

Friedrich Steimann

Objektorientierte Programmierung

Kurs 01814

19

Fakultät für
Mathematik und
Informatik



FernUniversität in Hagen

Das Werk ist urheberrechtlich geschützt. Die dadurch begründeten Rechte, insbesondere das Recht der Vervielfältigung und Verbreitung sowie der Übersetzung und des Nachdrucks, bleiben, auch bei nur auszugsweiser Verwertung, vorbehalten. Kein Teil des Werkes darf in irgendeiner Form (Druck, Fotokopie, Mikrofilm oder ein anderes Verfahren) ohne schriftliche Genehmigung der FernUniversität reproduziert oder unter Verwendung elektronischer Systeme verarbeitet, vervielfältigt oder verbreitet werden.

Inhaltsverzeichnis

Vorwort zur dritten Auflage.....	i
Zum Inhalt.....	i
Zur Form	iv
Vorkurs.....	1
Objektorientierte Programmierung	1
Objektorientierte Programmiersprachen	3
SMALLTALK	3
Warum gerade SMALLTALK?	5
Programmierung mit SMALLTALK	7
Verfügbare SMALLTALK-Systeme	8
Literatur zu SMALLTALK	9
Kurseinheit 1: Grundkonzepte der objektorientierten Programmierung	11
1 Objekte.....	11
1.1 Was ist ein Objekt?	12
1.2 Literale	13
1.3 Änderbarkeit von Objekten	15
1.4 Gleichheit und Identität von Objekten	16
1.5 Variablen.....	18
1.5.1 Inhalt.....	19
1.5.2 Sichtbarkeit.....	20
1.6 Zuweisung.....	21
1.7 Pseudovariablen.....	23
1.8 Aliasing	23
1.9 Lebenslauf von Objekten.....	25
1.10 Zusammenfassung	27
2 Beziehungen	27
2.1 Instanzvariablen	28
2.2 Unterscheidung von :1- und :n-Beziehungen	30
2.3 Teil-Ganzes-Beziehungen	32

2.4	Attribute	33
3	Zustand	34
3.1	Eingrenzung	34
3.2	Kapselung	36
4	Verhalten	36
4.1	Ausdrücke	37
4.1.1	Zuweisungsausdrücke	37
4.1.2	Nachrichtenausdrücke	37
4.1.3	Auswertung von Ausdrücken.....	41
4.1.4	Reihenfolge der Auswertung von geschachtelten Ausdrücken	43
4.2	Anweisungen	44
4.3	Methoden	45
4.3.1	Zuordnung von Methoden zu Objekten	48
4.3.2	Methodenaufruf und dynamisches Binden	49
4.3.3	Methoden als Parameter	52
4.3.4	Verbergen der Repräsentation des Zustands hinter Methoden	53
4.3.5	Operationen auf zustandslosen (unveränderlichen) Objekten	55
4.3.6	Konstante Methoden	56
4.3.7	Primitive Methoden.....	57
4.3.8	Protokoll	58
4.4	Blöcke	58
4.4.1	Home context und Closure	59
4.4.2	Continuation	60
4.4.3	Parametrisierte Blöcke	61
4.5	Kontrollstrukturen.....	62
4.5.1	Sequenz.....	62
4.5.2	Dynamisch gebundener Methodenaufruf	63
4.6	Abgeleitete Kontrollstrukturen	63
4.6.1	Verzweigung.....	64
4.6.2	Wiederholung	66
4.6.3	Iteration	66
4.6.4	Iterieren über <i>:n</i> -Beziehungen.....	68

5 Zusammenfassung der SMALLTALK-Syntax	70
6 Lösungen zu den Selbsttestaufgaben	71
Kurseinheit 2: Systematik der objektorientierten Programmierung 75	
7 Klassen	76
7.1 Klassifikation	77
7.2 Klassendefinitionen	78
7.3 Instanzierung	81
8 Metaklassen	83
8.1 Konstruktoren	86
8.2 Initialisierung	87
8.3 Factory-Methoden	91
8.4 Erzeugung von Klassen in SMALLTALK	92
8.5 Die Metaklassenleiter SMALLTALKS	93
8.6 Praktische Bedeutung der Metaklassen für die Programmierung	95
9 Generalisierung und Spezialisierung	96
9.1 Generalisierung	96
9.2 Spezialisierung	101
10 Vererbung und abstrakte Klassen	103
10.1 Vererbung	103
10.2 Vererbung in prototypenbasierten Sprachen	105
10.3 Abstrakte Klassen	106
11 Superklassen und Subklassen	110
11.1 Bedeutung der Subklassenbeziehung	111
11.2 Mechanismus der Vererbung von Superklassen auf Subklassen	112
11.3 Löschen von Methoden	112
11.4 Subklassenhierarchie und Vererbung unter Metaklassen	113
11.5 Dominanz der Vererbung	115
12 Dynamisches Binden	116
12.1 Nachrichten an <code>self</code>	118
12.2 Das Einbeziehen überschriebener Methoden: Nachrichten an <code>super</code>	120

12.3	Double dispatch.....	120
13	Programmieren mit Collections.....	122
13.1	Pflegen von $:n$ -Beziehungen	123
13.2	$:n$ -Beziehungen mit besonderen Eigenschaften	125
13.2.1	Dictionaries.....	125
13.2.2	Sortierte Collections	127
13.2.3	Arrays.....	129
13.3	Collections für andere Zwecke.....	129
14	Verhalten für alle Objekte.....	131
14.1	Kopieren von Objekten	131
14.2	Reinkarnation von Objekten	133
14.3	Kommunikation mit mehreren: Multicasting	134
14.4	Selbstdarstellung.....	136
15	Ein- und Ausgabeströme	137
16	Parallelität: aktive und passive Objekte	139
17	Lösungen zu den Selbsttestaufgaben	141
Kurseinheit 3: Typen in der objektorientierten Programmierung		147
18	Hintergrund	148
19	Deklaration, Definition und Verwendung von Programmelementen.....	153
20	Typdefinitionen und deren Verwendung.....	155
20.1	Induktiver Aufbau von Typen und Semantik.....	156
20.2	Verwendung definierter Typen	158
21	Zuweisungskompatibilität	159
22	Typäquivalenz.....	160
22.1	Strukturäquivalenz	161
22.2	Namensäquivalenz	163
23	Typerweiterung	164
24	Typkonformität.....	165

25 Typeinschränkung	167
26 Subtyping und Inklusionspolymorphie	171
26.1 Der Begriff des Subtyps	173
26.2 Strukturelles und nominales Subtyping	175
26.3 Kovarianz und Kontravarianz bei Methodenaufrufen	175
26.4 Inklusionspolymorphie.....	178
27 Typumwandlungen	179
28 Der Zusammenhang von Typen und Klassen	180
28.1 Subklassen und Subtypen.....	182
28.2 Typen als Schnittstellenspezifikationen von Klassen	182
28.3 Gründe für die Trennung von Typen und Klassen	183
29 Generische Typen oder parametrischer Polymorphismus	184
29.1 Einfacher parametrischer Polymorphismus	185
29.2 Collections als Standardanwendungsfall für parametrischen Polymorphismus	187
29.3 Parametrischer Polymorphismus und Inklusionspolymorphie	188
29.4 Beschränkter parametrischer Polymorphismus.....	192
29.5 Rekursiv beschränkter parametrischer Polymorphismus.....	193
30 Parametrischer Polymorphismus und das Kovarianzproblem	195
31 Grenzen der Typisierung	196
32 Fazit.....	197
33 Lösungen zu den Selbsttestaufgaben	197
Kurseinheit 4: JAVA	199
34 Das Programmiermodell JAVAs.....	199
35 Objekte und Typen	201
35.1 Literale	201
35.2 Gleichheit und Identität.....	202
35.3 Variablen und Zuweisungen	202
35.4 Operationen und Methoden.....	203
35.5 Zuweisungskompatibilität.....	204

36 Klassen	205
36.1 Klassendefinitionen	205
36.2 Subklassenbeziehung	206
36.3 Konstruktoren.....	208
36.4 Überschreiben, Überladen und dynamisches Binden.....	209
37 Ausdrücke	211
38 Anweisungen, Blöcke und Kontrollstrukturen.....	213
39 Module	217
39.1 Klassen und Pakete als Module.....	217
39.2 Abhängigkeiten zwischen Modulen.....	219
39.3 Die Module von JAVA 9	220
40 Interfaces	221
40.1 Interfaces als Schnittstellen.....	222
40.2 Interfaces als abstrakte Klassen.....	223
41 Arrays.....	224
42 Aufzählungstypen	227
43 Generische Typen	227
43.1 Einfache parametrische Typdefinitionen	227
43.2 Parametrische Typen und Subtyping: Wildcards.....	230
43.3 Beschränkter parametrischer Polymorphismus in JAVA	234
43.4 Rekursiv beschränkter parametrischer Polymorphismus.....	235
43.5 Generische Methoden.....	236
43.6 Generische Variablen	237
44 Dynamische Typprüfung in JAVA.....	237
44.1 Type casts.....	238
44.2 Typtests	239
45 Programmieren mit Interfaces	239
46 Interne und externe Iteration über Collections.....	242
46.1 Externe Iteration	242

46.2	Interne Iteration	243
47	Spezielle Klassen.....	244
47.1	Object	245
47.2	Exception handling	245
47.3	Multi-threading.....	248
47.4	Metaprogrammierung.....	249
48	Ein abschließendes Beispiel.....	249
49	Lösungen zu den Selbsttestaufgaben	252
Kurseinheit 5: Andere objektorientierte Programmiersprachen.....		255
50	C#.....	255
50.1	Das Programmiermodell von C#	256
50.2	Gemeinsamkeiten mit und kleinere Unterschiede zu JAVA.....	257
50.3	Zusätzliche Ingredienzien von Klassendefinitionen in C#	259
50.3.1	Properties	259
50.3.2	Indexer	261
50.3.3	Ereignisse (Events).....	261
50.4	Das Typsystem von C#	262
50.4.1	Die Typhierarchie von C#.....	262
50.4.2	Interfacetypen in C#.....	264
50.4.3	Generizität in C#	266
50.4.4	Die dynamische Komponente	267
50.5	Ausblick.....	268
51	C++	268
51.1	Das Programmiermodell von C++	269
51.2	Klassen	269
51.3	Friends	271
51.4	Mehrfachvererbung	272
51.5	Das Typsystem von C++	273
51.5.1	Statische Komponente	273
51.5.2	Dynamische Komponente.....	275
51.6	Der ganze Rest.....	276

52 EIFFEL.....	276
52.1 Das Programmiermodell EIFFELS	277
52.2 Klassen als Module	278
52.3 Anweisungen	279
52.4 Vererbung und Überladen.....	279
52.5 Das Typsystem EIFFELS	280
52.5.1 Ein motivierendes Beispiel.....	281
52.5.2 Statische Komponente	283
52.5.3 Die dynamische Komponente	288
52.6 Zusicherungen in EIFFEL: Vorbedingungen, Nachbedingungen und Klasseninvarianten.....	290
52.7 Tupel anstelle von Klassen.....	291
52.8 Fazit	291
53 Lösungen zu den Selbsttestaufgaben	292
Kurseinheit 6: Probleme der objektorientierten Programmierung	293
54 Das Problem der Substituierbarkeit.....	293
54.1 Der Begriff der Substituierbarkeit.....	294
54.2 Subtyping und das Prinzip der Substituierbarkeit	296
54.3 Verhaltensbasiertes Subtyping	298
54.4 Das Liskov-Substitutionsprinzip	300
54.5 Relativität der Substituierbarkeit	303
55 Das Fragile-base-class-Problem	304
56 Das Problem der schlechten Tracebarkeit.....	309
57 Das Problem der eindimensionalen Strukturierung.....	314
58 Das Problem der mangelnden Kapselung	315
59 Das Problem der mangelnden Skalierbarkeit.....	319
60 Das Problem der mangelnden Eignung	320
61 Lösungen zu den Selbsttestaufgaben	323
Kurseinheit 7: Objektorientierter Stil	325

62	Namenwahl.....	327
62.1	Verwendung von Abkürzungen.....	328
62.2	Namenskonventionen	328
62.2.1	Mechanische (syntaktische) Namenskonventionen.....	329
62.2.2	Grammatikalisch-inhaltliche (semantische) Namenskonventionen.....	329
63	Formatierungskonventionen.....	331
64	Kurze Methoden.....	332
65	Deklarativer Stil	333
66	Der Bibliotheksgedanke	334
67	Ausgewogene Verteilung	335
68	Das Gesetz Demeters (Law of Demeter).....	336
68.1	Bedeutung.....	336
68.2	Automatische Überprüfung	337
68.3	Ein Beispiel	338
69	Klassenhierarchie.....	339
70	Fazit.....	341
71	Lösungen zu den Selbsttestaufgaben	341
	Verzeichnis der Weblinks im Rand	343
	Index	347

Vorwort zur dritten Auflage

Seit der ersten Auflage dieses Kurses im Wintersemester 2006/2007 hat sich viel getan: JAVA als vormals dominierende objektorientierte Programmiersprache hat weiter Grund u. a. an C# verloren und beiden wird zunehmend von JAVASCRIPT der Rang abgelaufen. Alle drei sind nicht nur objektorientiert, sondern haben (mittlerweile) auch starke funktionale Sprachanteile (die sog. Lambda-Ausdrücke), die im Vergleich zur imperativen Programmierung wesentlich kompaktere Programme ermöglichen. JAVA hat zudem seit der Übernahme durch ORACLE viel von seinem Community-Charakter eingebüßt; die zögerliche Weiterentwicklung der Sprache hat JAVA-basierten Konkurrenten wie SCALA oder KOTLIN Raum für Entwicklung gegeben. Ironischerweise sind Vorbild für manche der wichtigsten Weiterentwicklungen alte objektorientierte Programmiersprachen wie SMALLTALK oder EIFFEL; beide spielen in der aktuellen Programmierpraxis zwar nur noch eine Nebenrolle, ihr Erbe prägt aber die heutige Programmierung maßgeblich.

Zum Inhalt

Dieser Kurs stellt die objektorientierte Programmierung so dar, dass ein Gegensatz zur klassischen imperativen Programmierung entsteht. Um diesen Gegensatz wahrzunehmen, müssen Sie natürlich die imperative Programmierung kennen, also z. B. den Kurs 01613 „Einführung in die imperative Programmierung“ absolviert haben. Gleichwohl ist dieses Wissen keine Voraussetzung — Sie können mit dem Ihnen vorliegenden Kurs die objektorientierte Programmierung auch als erstes lernen.



So sind die formalen Voraussetzungen zur erfolgreichen Bearbeitung dieses Kurses zunächst gering. Insbesondere bringt es Ihnen wenig, wenn Sie schon „Java können“ — im ungünstigsten Fall müssen Sie sogar erst einiges vergessen, um die ersten Kurseinheiten unvoreingenommen verinnerlichen zu können. Wichtig für den Erfolg ist jedoch die grundsätzliche Bereitschaft, sich mit der Programmierung und deren Werkzeugen auseinanderzusetzen — insbesondere wird Ihnen der Kurs nur wenig bringen, wenn Sie nicht bereit sind, die damit verbundenen Programmierübungen wirklich durchzuführen. Wenn alles gutgeht, sollten Ihnen diese Übungen allerdings auch Freude bereiten.

Wenn Sie diesen Kurs erfolgreich absolviert haben, dann sollten Sie das folgende können:

- objektorientiert denken,
- objektorientiert programmieren,
- die Bestandteile objektorientierter Programmiersprachen erkennen und dabei insbesondere die Bedeutung eines Typsystems richtig einschätzen,
- die Schwächen der objektorientierten Programmierung benennen,

Ziele

- Kriterien für die Auswahl einer für ein bestimmtes Projekt geeigneten Programmiersprache aufstellen sowie
- Aussagen zum eigenen und zum Programmierstil anderer machen.

Dem ersten Ziel kommt übrigens eine größere Bedeutung bei, als man bei oberflächlicher Betrachtung annehmen würde: Nur wer objektorientiert denkt, schreibt Programme, die das Attribut „objektorientiert“ verdienen. Zwar tun die pureren unter den Vertretern der objektorientierten Programmiersprachen einiges, um ihren Programmiererinnen einen objektorientierten Stil aufzuzwingen, aber wer nicht will, kann diese Schubse auch ignorieren. Wer hingegen objektorientiert in Sprachen programmieren soll, die selbst gar nicht unbedingt objektorientiert sind, die wird enorme Schwierigkeiten bekommen, wenn sie nicht wenigstens objektorientiert denken kann. Und nicht zuletzt ist das objektorientierte Denken Voraussetzung für so manch andere Aktivität im Softwareerstellungsprozess, die u. U. noch weit von der Programmierung entfernt ist: So findet die Modellierung von Softwaresystemen heute meistens mit Modellierungssprachen wie der *Unified Modeling Language (UML)* statt, die als objektorientiert gelten. Wenn Sie also nicht objektorientiert denken, dann können Sie auch mit solchen Modellen wenig anfangen und werden zu einem entsprechenden Prozess nicht wirklich beitragen können.

Ein universitärer Kurs zur objektorientierten Programmierung kann kein Programmierkurs sein. Er muss sich vielmehr auf die Konzepte der objektorientierten Programmierung konzentrieren und sie im Kontext darstellen, ganz so wie Kurse zur imperativen, logischen oder funktionalen Programmierung dies auch tun müssen. Programmieren lernt man ohnehin am besten in der Praxis; ein Kurs wie dieser soll allerdings helfen, dass dieser Lernprozess informiert, ohne Um- und Irrwege, verläuft.

kein

Programmierkurs!

Der Vorkurs bringt einen kurzen praktischen Einstieg in die objektorientierte Programmierung, der Ihnen beim Verständnis des Hauptkurses helfen soll. Wichtig für Sie ist hier vor allem, dass Sie sich, wie in den Übungen gefordert, mindestens eine SMALLTALK-Umgebung besorgen und mit dieser vertraut machen. Das Verständnis von weiten Teilen der folgenden Kurseinheiten hängt entscheidend davon ab, ob Sie in der Lage sind, den Stoff praktisch nachzuvollziehen und Ihr Erlerntes an eigenen kleinen Experimenten zu überprüfen. Programmieren ist eine praktische Tätigkeit; es muss geübt werden wie das Feilen in einer Eisenwerkstatt und wer nur glaubt, sie hätte alles verstanden, kann noch lange nicht programmieren. Objektorientiertes Programmieren ist davon keine Ausnahme.

die Kurseinheiten im einzelnen

Kurseinheit 1 beginnt mit einer ausführlichen Einführung der wenigen Grundkonzepte der objektorientierten Programmierung. Sie behandelt im wesentlichen Objekte, Variablen, Beziehungen zwischen Objekten, Nachrichten, die zwischen Objekten ausgetauscht werden können, sowie Methoden, die die Reaktion auf den Empfang einer Nachricht festlegen. Auf die Definition von Objekten als Konglomerat von Variablen und Methoden sowie auf die Erzeugung von Objekten, die nicht „schon da“ sind, wird hier nicht eingegangen. Diese



strikte Trennung erlaubt es, den Klassenbegriff, der für die meisten objektorientierten Programmiersprachen eine zentrale Rolle spielt, als einen Freiheitsgrad objektorientierter Programmierung darzustellen.

In Kurseinheit 2 wird dann die Klasse als struktur- und ordnungsstiftendes Konstrukt der objektorientierten Programmierung vorgestellt. Eng mit dem Klassenbegriff verbunden sind die Instanziierung von Klassen zu Objekten, der Übergang von Klassen zu ihren Metaklassen sowie die Vererbung zwischen Klassen. Insbesondere die Vererbung hat in der objektorientierten Programmierung eine wechselhafte Geschichte hinter sich; auf die mit ihr verbundenen Probleme wird jedoch erst in den folgenden Kurseinheiten eingegangen. Stattdessen werden in Kurseinheit 2 noch das dynamische Binden und seine Bedeutung als universelle Kontrollstruktur dargestellt.

Mit Kurseinheit 3 wird in das Thema Typsysteme für die objektorientierte Programmierung eingeführt. Dabei wird ausgenutzt, dass SMALLTALK kein Typsystem hat; Typsysteme können hier also ohne Vorbelastung durch eine spezielle Sprache auf die objektorientierte Programmierung aufgepropft werden. Dabei wird insbesondere auf den für die objektorientierte Programmierung so zentralen Begriff des Subtyping hingearbeitet; aber auch die generischen Typen, die spätestens seit JAVA 5 und C# 2.0 auch zum täglichen Brot kommerzieller Programmiererinnen gehören dürften, werden so eingeführt. Der dabei zum Einsatz kommende SMALLTALK-Nachfolger STRONGTALK wird dazu als Vorlage genommen, aber nur insoweit, als sich die Eigenschaften dessen Typsystems auch in anderen Programmiersprachen wiederfinden.

Ab Kurseinheit 4 werden dann andere Programmiersprachen vorgestellt. Kurseinheit 4 zollt zunächst JAVA als der heute wohl immer noch wichtigsten objektorientierten Programmiersprache Tribut: Die Präsentation orientiert sich dabei an der Einführung in die objektorientierte Programmierung aus den vorangegangenen Kurseinheiten. Dies erlaubt eine prägnante Darstellung der wesentlichen Konzepte und Unterschiede (und führt zu einer eher ungewöhnlichen Sicht auf JAVA als Programmiersprache). In Kurseinheit 5 werden dann auf analoge Weise der Reihe nach C#, C++ und EIFEL präsentiert. Dies soll nicht zuletzt Ihrem Verständnis von der Vielfalt der Sprachenlandschaft dienen und Ihnen die Einsicht vermitteln, dass die guten Seiten der objektorientierten Programmierung nicht in einer Sprache allein zu finden sind.

Kurseinheit 6 thematisiert die wichtigsten Probleme der objektorientierten Programmierung. Es sind dies im wesentlichen die des Subtyping und der damit verbundenen Substituierbarkeit von Objekten sowie der Kapselung von Objekten zu größeren Einheiten, die ihr Inneres verbergen. Beide scheinen auf den ersten Blick gelöste Probleme zu sein; tatsächlich sind sie es aber keineswegs. Auch auf Probleme der Skalierbarkeit der objektorientierten Programmierung wird hier eingegangen.

In Kurseinheit 7 lassen wir schließlich die Konstrukte objektorientierter Programmiersprachen hinter uns und gehen ausführlich auf das ein, was man gemeinhin objektorientierten



Stil nennt. Der Programmierstil hat sich nämlich seit den Anfängen der prozeduralen Programmierung mit Sprachen wie PASCAL, wie er von Größen wie EDSGER DIJKSTRA, TONY HOARE oder NIKLAUS WIRTH gelehrt wurde, mindestens so sehr geändert wie die Sprachen selbst. Objektorientierte Sprachen zeichnen sich dadurch aus, wie gut sie objektorientierten Stil umzusetzen erlauben, und Sie als Programmiererinnen, wie Sie ihn umzusetzen in der Lage sind.

Zur Form

Dieser Kurs verwendet das aktuelle Layout der Fernuniversität, mit einigen Erweiterungen.

Hinter den Piktogrammen im Rand verbergen sich Links auf Online-Quellen. Sie ersetzen konventionelle Quellenangaben und Hinweise auf weiterführende Literatur gleichermaßen. Auf verlinkte Artikel digitaler Bibliotheken sollten Sie aus dem Netz der Fernuni zugreifen können; evtl. müssen Sie dafür das VPN der Fernuni nutzen. Ein Verzeichnis aller verlinkten Quellen finden Sie am Ende des Kurstextes.

Piktogramme im Rand

Link



Neben der üblichen Verlinkung von Querverweisen sind im Text auch Begegriffsanführungen mit ihren -definitionen verknüpft, soweit letztere explizit vorhanden sind. Sie erkennen Anführungen am kursiven und Definitionen am fetten Schriftsatz. Alle übrigen Begegriffsanführungen und die Definitionen selbst verweisen in den Index; dessen Seitenzahlen verweisen wiederum auf die Vorkommen der Begriffe im Text. Prost!

Links im Text

Der Barcode in den Fußzeilen des Hauptteils dieses Kurstextes codiert die Kursnummer, eine zweistellige Variantennummer sowie eine dreistellige, absolute Seitenzahl. Er dient meiner **Papier/Digital-Brücke**, einer experimentellen Android-App, die sie sich unter nebenstehendem Barcode (www.feu.de/pdb/) herunterladen können. Mit Hilfe der Papier/Digital-Brücke können Sie alle Links des PDFs auch in der Papierversion direkt nutzen, sich den Kurstext in erweiterter Form seitenweise vorlesen lassen oder im Studienbrief notierte Anmerkung mit anderen teilen. Die App befindet sich noch im Experimentierstadium; ob sie weiterentwickelt wird, hängt auch von Ihrer Rückmeldung ab.

Barcode im Seitenfuß



Gemäß Vorgabe der Fernuni ist dieser Kurstext genderisiert. Da die konsequente Verwendung geschlechtsneutraler Formulierungen die persönliche Ansprache verwässert, habe ich mich entschlossen, zwei Varianten zu erstellen: eine, die das generische Femininum bemüht (die vorliegende) und eine, die das Maskulinum verwendet. Da Kursbelegung und Materialversand an der Fernuni derzeit keine Varianten unterstützen, bekommen Sie die erste Variante gedruckt und geliefert — sofern Sie die zweite bevorzugen, können Sie sich das entsprechende PDF aus der VU herunterladen. Falls Sie wissen wollen, wie groß der Unterschied zwischen den beiden Varianten ist: Es sind aktuell 285 Phrasen geschlechtsspezifisch (wobei ich sicher noch einige übersehen habe).

Genderisierung

Hagen, im Januar 2020



Vorkurs

In diesem Vorkurs werden die praktischen Voraussetzungen für die folgenden Kurseinheiten geschaffen. Sie werden dazu an eine objektorientierte Programmiersprache, SMALLTALK, und die damit verbundene Programmierumgebung herangeführt. Es ist für den weiteren Verlauf wichtig, dass Sie mit dem Umgang mit mindestens einem SMALLTALK-System vertraut sind, auch wenn Sie noch nicht verstehen, wie es funktioniert und warum es so anders zu sein scheint als alles, was Sie vielleicht schon kennen.

Parallel zum Lesen dieses Vorkurses, der ausgesprochen leichte Kost ist, sollten Sie ein oder mehrere SMALLTALK-Systeme installieren. Wie das geht, finden Sie in den Übungsaufgaben zu diesem Vorkurs. Die Aufgaben sollten Sie unbedingt durchführen und ggf. so oft wiederholen, bis Sie die Anleitung dazu nicht mehr brauchen. Es wäre ärgerlich, wenn Sie die Einsendeaufgaben der folgenden Kurseinheiten nur deswegen nicht lösen können, weil Sie mit Ihrem SMALLTALK-System nicht zurechtkommen.

Objektorientierte Programmierung

Das Adjektiv „objektorientiert“ wurde nach eigenem Bekunden von ALAN KAY geprägt:

*... a new design paradigm — which I called object-oriented — for attacking
large problems of the professional programmer ...*



KAY erhielt 2003 den renommierten Turing-Award (auch als Nobelpreis der Informatik bezeichnet), und zwar u. a. für seine Rolle bei der Entwicklung von SMALLTALK:

*For pioneering many of the ideas at the root of contemporary object-oriented
programming languages, leading the team that developed SMALLTALK, and for
fundamental contributions to personal computing.*

Hütern der englischen Sprache, die nichts mit Programmieren zu tun haben, stößt der Begriff „object-oriented“ übrigens sauer auf, denn sprachlich korrekt müsste es „object-orientated“ heißen. Zum Glück wurde dieser Frevel, der oft Skandinavierinnen zugeschrieben wurden, nach eigenem Bekunden von einem Muttersprachler begangen.



Das der objektorientierten Programmierung zugrundeliegende Weltbild ist das von *Objekten*, die jeweils eine *Identität* haben, die einander *Nachrichten* schicken und die in Reaktion auf die Nachrichten durch Anweisungen, die in *Methoden* festgehalten sind, ihren *Zustand* verändern. Welche Nachrichten ein Objekt versteht, zählt zu seinen *Eigenschaften*. Objekte können zudem entstehen und wieder vergehen — das objektorientierte Weltbild ist also in vielerlei Hinsicht dynamisch.

Damit sich Objekte gezielt Nachrichten schicken können, müssen sie sich kennen. Welche anderen Objekte ein Objekt kennen kann, zählt zu seinen Eigenschaften; welche es kennt, macht den Zustand eines Objektes aus und unterliegt mit diesem der Veränderung. Um neue Bekanntschaften zu schließen, können einem Objekt ein oder mehrere andere Objekte als Parameter einer Nachricht geschickt werden. Der Empfang einer Nachricht durch ein Objekt führt in der Regel zum Versand weiterer Nachrichten durch das empfangende Objekt sowohl an andere Objekte als auch an sich selbst. Auch das Entstehen und Vergehen von Objekten erfolgt in der Regel als Reaktion auf den Empfang einer Nachricht.

Da Objekte so allgemeine Dinge wie Personen oder Dokumente, aber auch so spezielle Dinge wie Zahlen oder Wahrheitswerte sein können und Nachrichten so allgemeine wie „schreib dich ein“ oder „drucke dich aus“, aber auch so spezielle wie „+“ oder „-“, hat man damit schon fast alles, was man zum Programmieren braucht. Die einzigen zusätzlich benötigten Konzepte sind das der *Variable* und der *Wertzuweisung*, die Sie vermutlich bereits aus anderen Programmiersprachen kennen.

**Vererbung bzw.
Subtyping**

Der Begriff der Objektorientierung verlangt aber noch mindestens eine weitere Eigenschaft, die der *Vererbung*. Sie besagt, dass Objekte Eigenschaften von anderen erben können, dass also insbesondere nicht jedes Objekt für sich seine Eigenschaften definieren muss. Allerdings hat sich der Begriff der Vererbung als ein zwar leicht zu fassender, aber ungemein schwierig umzusetzender erwiesen, zumindest, wenn man sich durch Vererbung nicht in dem Umfang Schwierigkeiten einhandeln will, in dem man sich Arbeit erspart. Und so sehen viele heute eher das sog. *Subtyping* anstelle der Vererbung als für die objektorientierte Programmierung wesensbildende Eigenschaft.

Als die objektorientierte Programmierung noch angepriesen werden musste, wurden immer wieder dieselben Vorteile genannt:

**Eigenschaften der
objektorientierten
Programmierung**

- einfache *Wiederverwendung*,
- natürliche *Modularisierung* sowie
- *Durchgängigkeit* der Konzepte von der Analyse über das Design (beide heute gern unter dem Begriff „Modellierung“ zusammengefasst) bis hin zur Implementierung.

Auch wenn kein kausaler Zusammenhang bestehen sollte: Seit sich die objektorientierte Programmierung durchgesetzt hat, redet kaum eine noch von der *Softwarekrise*.



Objektorientierte Programmiersprachen

Inzwischen gibt es eine große Anzahl von Programmiersprachen, die objektorientiert sind. Viele ältere Programmiersprachen wie z. B. PASCAL oder ADA sind in Fortschreibungen wie OBERON oder ADA-95 objektorientiert geworden, aber auch jüngere Sprachen wie z. B. PHP haben sich nach und nach „objekt-orientiert“. Nicht mehr wegzudenken aus der modernen Softwareentwicklung ist die Objektorientierung jedoch erst seit Einführung der von C und SMALLTALK abgeleiteten objektorientierten Sprachen C++, JAVA und C#: Diese haben die kommerzielle Softwareentwicklung quasi im Sturm erobert.

Zu den bekannteren objektorientierten Programmiersprachen (oder zumindest Programmiersprachen mit objektorientierten Anteilen) zählen heute (in alphabetischer Reihenfolge): ADA 95, BETA, C++, C#, COMMON LISP, D, OBJECT PASCAL (BORLAND DELPHI), DYLAN, EIFFEL, SATHER, FORTRAN 2003, JAVA, JAVASCRIPT, MODULA-3, OBERON, OBJECTIVE-C, PERL 5, PHP 5, SELF, PYTHON, RUBY, SCALA, SIMULA, SMALLTALK, SELF und VISUAL BASIC. Diese haben teilweise ihre Wurzeln in der prozeduralen Programmierung (mit Programmiersprachen wie ALGOL, PASCAL oder C), teilweise in der funktionalen (mit Programmiersprachen wie LISP); aber auch ganz eigenständige wie beispielsweise BETA sind darunter. Auch wenn nicht alle Sprachen gleich praxisrelevant sind, kann es sich doch lohnen, sie zu kennen, denn jede hat ihre ganz eigenen Stärken und Schwächen und auch wenn man in einer anderen Sprache programmieren muss, kann man doch vielleicht die eine oder andere Idee aus einer anderen Sprache emulieren.

bekannte
objektorientierte
Programmier-
sprachen



WIKIPEDIA

SMALLTALK

SMALLTALK wurde seit dem Beginn der 1970er Jahre bei der Firma Xerox in Palo Alto, USA, (genauer: im Palo Alto Research Center PARC) entwickelt, wurde dort 1983 unter der Bezeichnung SMALLTALK-80 der Öffentlichkeit präsentiert und gilt bis heute als die objektorientierteste objektorientierte Programmiersprache. Auch wenn ALAN KAY in seinem Rückblick ein anderes Bild zeichnet, wird als einzige direkte Vorgängerin SMALLTALKS eigentlich immer die Programmiersprache SIMULA genannt, die bereits ca. zehn Jahre früher, also in den 1960er Jahren, von den beiden Norwegern OLE-JOHAN DAHL und KRISTEN NYGAARD (die dafür schon vor KAY im Jahr 2001 den Turing-Award erhielten und nach denen inzwischen selbst ein Preis benannt ist) an der Universität Oslo konzipiert wurde und die unter dem Namen SIMULA-67 in Europa (aber wohl auch nur dort) einige Verbreitung gefunden hatte. Ein anderer wichtiger Einfluss auf SMALLTALK war aber sicher LISP, denn gewisse *funktionale* Eigenschaften (also die Tatsache, dass in der Sprache Funktionsausdrücke ausgewertet werden) kann und will SMALLTALK nicht verbergen.



WIKIPEDIA



WIKIPEDIA

Die Grundidee SMALLTALKS lässt sich in einem einzigen, einfachen Satz zusammenfassen:

Grundidee
SMALLTALKS

Alles ist ein Objekt.



Wie Sie noch sehen werden, ist mit „alles“ wirklich fast alles gemeint: Klassen, Instanzen, Variablen, Nachrichten und Methoden sind neben anderen Dingen wie Browsern, Editoren, dem Compiler, dem Dateisystem und jeder Änderung, die Sie machen, alles Objekte. Auch kennt SMALLTALK keine primitiven Werte: Selbst so grundlegende Dinge wie Zahlen und die Booleschen Wahrheitswerte `true` und `false` sind Objekte, die grundsätzlich nicht anders behandelt werden als etwa ein Dokument oder eine Grafik. Dadurch bedingt kommt SMALLTALK mit einer minimalen Menge von Konzepten aus, was der Sprache eine gewisse Eleganz verleiht. Das Sparsamkeitsprinzip wird dadurch auf die Spitze getrieben, dass viele der von anderen Programmiersprachen bekannten Konstrukte in SMALLTALK nicht primitiv (Bestandteil der Sprachdefinition) sind, sondern darin ausgedrückt (emuliert) werden. Die Kargheit der Sprache wird denn auch durch eine umfangreiche Klassenbibliothek, die dadurch gewissermaßen Bestandteil der Sprachdefinition wird, kompensiert, so dass einer trotz der Andersartigkeit vieles bei der Programmierung in SMALLTALK vertraut vorkommt. Es ist sogar so, dass in SMALLTALK viele der bekannten Programmierkonstrukte Seite an Seite mit solchen stehen, die man schon immer gern gehabt hätte, die andere Programmiersprachen aber (noch) nicht anbieten, und dass der bereits vorhandene Vorrat mit relativ wenig Aufwand von einer selbst um weitere Konstrukte ergänzt werden kann.

SMALLTALK ist aber mehr als nur eine Programmiersprache: Es ist zugleich ein Betriebssystem und eine vollständig integrierte Entwicklungsumgebung. Jede mit SMALLTALK entwickelte Anwendung ist dabei lediglich eine Änderung des SMALLTALK-Systems, mit dem sie entwickelt wurde. Die Programmiererin macht also nichts weiter, als ihre SMALLTALK-Installation solange zu verändern, bis sie die Funktionalität der gewünschten Anwendung hat. Dazu kann sie nicht nur Funktionen hinzufügen, sondern auch existierende verändern oder entfernen. Die Programmierung in SMALLTALK ist dabei vollkommen unabhängig von so primitiven Konzepten wie Datei oder Verzeichnis, die in anderen Sprachen Teil der Definition sind (oder zumindest diese maßgeblich beeinflussen).¹ Auch die üblichen, langwierigen Editier-Speicher-Kompilier-Test-Zyklen gibt es nicht — Sie arbeiten immer an einem laufenden System (die sog. *Live-Programmierung*). SMALLTALK ist ein experimentelles System im besten Sinne und Programmieren in SMALLTALK ist mit seinen extrem kurzen Editier-Kompilier-Ausprobier-Zyklen eine höchst dynamische Angelegenheit.

SMALLTALK ist mehr

Wer kommerzielle Softwareentwicklung aus eigener Anschauung kennt, ahnt vielleicht, dass es mit dem praktischen Einsatz von SMALLTALK Probleme geben könnte: Die Philosophie, keine Auslieferung von binären, d. h. im wesentlichen nicht durch die Benutzerin veränderbaren, Programmen vorzusehen, mutet aus kommerzieller Sicht etwas eigenständlich an. Zudem war die gegenüber nativ kompilierten Programmen geringere Performance (Ausführungsgeschwindigkeit) sicher ein Problem. Dennoch setzte Mitte der neunziger Jahre ein SMALLTALK-Boom ein: Vor allem Banken und andere

kommerzielle Verwendung

¹ Diese Verbannung des Begriffs der Datei aus der Definition der Programmiersprache findet man auch in JAVA, wo er durch den Begriff der *Compilation unit* ersetzt wird. IBMs Entwicklungsumgebung VISUALAGE FOR JAVA (die der Vorläufer von ECLIPSE war) speicherte Programme auch zunächst in einer Datenbank; mit dem Übergang zu ECLIPSE ging man aber wieder auf Dateien über, schon weil viele andere Programmierwerkzeuge (darunter Versionskontrollsysteme) auf Dateibasis arbeiten.



Unternehmen mit großen betrieblichen Informationssystemen begannen, erste hausinterne Applikationen mit SMALLTALK zu entwickeln.² Geschätzt wurde vor allem die (im Vergleich zu den branchenüblichen Sprachen wie COBOL, aber auch zu sog. höheren Programmiersprachen wie MODULA, C oder C++) sehr viel höhere Produktivität. Ironischerweise wurde dieses zarte Pflänzchen wenig später ausgerechnet von JAVA, einer Programmiersprache, die sich manches von SMALLTALK abgeguckt hat, in Sachen Laufzeit um keinen Deut besser ist, in Sachen Produktivität jedoch trotz aller Neuerungen heute immer noch nicht an SMALLTALK heranreicht, plattgewalzt. Während der kurzen Blütezeit von SMALLTALK entwickelte IBM übrigens seine Entwicklungsumgebung VISUALAGE, die wenig später (und noch in SMALLTALK programmiert) für JAVA adaptiert wurde und aus der das heutige ECLIPSE-Projekt hervorgegangen ist. Doch während ECLIPSE als Entwicklungsumgebung für JAVA das SMALLTALK-System, aus dem es hervorgegangen ist, seit langem weit übertrifft, hat die Programmiersprache JAVA bis heute gebraucht, um an die Produktivität von SMALLTALK heranzureichen (machen würden sagen: Sie hat es bis heute nicht geschafft).

Vielleicht das größte Problem SMALLTALKS war aber, dass es zu früh kam. Seine Architektur, die am unteren Ende auf einer virtuellen Maschine (VM) mit automatischer Speicherbereinigung (*Garbage collection*) beruhte und am oberen auf einer Bedienoberfläche mit pixelgraphischen, überlappenden Fenstern und Zeigegeräten wie Maus oder Tablet, stellte recht deftige Anforderungen an die Hardware, die damals noch klobig und teuer war. Außerdem wurden fast alle innovativen Konzepte SMALLTALKS quasi auf der grünen Wiese entwickelt und mussten daher erst einmal in der Praxis erprobt und über einen langen Zeitraum verbessert werden, bis sie wirklich zu gebrauchen waren. Am Ende dieser Entwicklung haben dann nur wenige das Potential erkannt, das in SMALLTALK steckte, darunter aber immerhin Steve Jobs, dessen Firma Macintosh-System ganz wesentliche Elemente SMALLTALKS (Maus, überlappende Fenster) übernommen und zur Verbreitung geführt hat.

**Quell zahlreicher
Innovationen**

Warum gerade SMALLTALK?

Man kann lange darüber diskutieren, welche der zahlreichen objektorientierten Programmiersprachen am besten für das Erlernen der objektorientierten Programmierung im Rahmen der universitären Lehre geeignet ist. In dem Ihnen vorliegenden Kurs habe ich mich vor allem aus didaktischen Gründen für SMALLTALK entschieden. Dabei soll der Kurs kein SMALLTALK-Kurs sein; Sie sollen vielmehr die objektorientierten Konzepte und damit auch das objektorientierte Denken verinnerlichen, und SMALLTALK scheint mir dafür am besten geeignet. Gleichwohl setzt das Verinnerlichen einiges an Übung „am Objekt“ voraus, und deswegen

² Das oft zitierte C3-Projekt, mit dem die Disziplin des Extreme Programming begründet wurde, war ein SMALLTALK-Projekt. Leider ist mir keine Arbeit bekannt, die untersucht hätte, auf welche Faktoren genau sich der Erfolg dieses Projekts zurückführen lässt — man kann aber wohl davon ausgehen, dass mit Kent Beck, Ward Cunningham, Ron Jeffries und Martin Fowler einige der bekanntesten (und wohl auch besten) Programmierer der Zeit mit an Bord waren, die nun einmal alle SMALLTALK sprachen.



haben die ersten Kurseinheiten durchaus den Charakter einer SMALLTALK-Einführung. Spätestens ab Kurseinheit 4 wird jedoch der Vorhang gehoben und der Blick auf die Weiten der heutigen objektorientierten Programmiersprachenlandschaft freigegeben.

SMALLTALK ist von Anfang an als eine leicht zu erlernende Sprache konzipiert worden. Insbesondere erweist sich als didaktischer Vorteil, dass man zum Erlernen der Sprache keine Programme schreiben, also genau genommen gar nicht programmieren muss — stattdessen führt man mit dem System einfach „Smalltalk“. Dazu kommt, dass die Grammatik der Sprache ausgesprochen klein ausfällt: Es gibt keine Schlüsselwörter, nur wenige (fünf!) reservierte Namen (JAVA hingegen hat 50 Schlüsselwörter und PASCAL immerhin auch schon 35) und lediglich die aus dem normalen Schriftgebrauch bekannten Interpunktionszeichen haben eine durch die Sprachdefinition festgelegte Bedeutung. So reicht in SMALLTALK für „Hello World!“ bereits der einfache Ausdruck

¹ Transcript show: 'Hello World!'

um den Text auf dem Ausgabeterminal des SMALLTALK-Systems, Transcript genannt, auszugeben. Dabei handelt es sich jedoch nicht um ein Programm im landläufigen Sinne, sondern lediglich um einen Ausdruck, der in der Entwicklungsumgebung ausgewertet („ausgeführt“) werden kann und als Ergebnis die Ausgabe der Zeichenkette „Hello World!“ bewirkt.³ Tatsächlich gibt es in SMALLTALK wie bereits erwähnt so etwas wie ein Programm, das Sie schreiben, eigentlich gar nicht; stattdessen handelt es sich beim SMALLTALK-System selbst um ein (laufendes) Programm, dass Sie als Programmiererin — während es läuft — so verändern, dass es neben allen anderen Dingen, die es kann, auch das tut, was Sie wollen (das bereits erwähnte *Live programming*). Je nach zu lösendem Problem kann Ihr eigener Beitrag dabei aus einem einzigen Ausdruck bestehen oder die Definition einer Vielzahl neuer Klassen umfassen. Die Werkzeuge, die Sie dabei benutzen, sind selbst auf diese Weise entstanden und können Teil Ihrer Anwendung sein.

Diese Besonderheit des SMALLTALK-Systems ist aber nicht nur ein faszinierender Denkansatz, sondern ermöglicht auch eine sehr praktische Herangehensweise bei der Vermittlung der Sprache. Es ist nämlich möglich, die Sprache durch Ausprobieren zu erlernen, ohne sich vorher Gedanken über Dinge wie Editoren, Dateien oder gar Verzeichnisse, den Aufruf des Compilers oder den Start des Programms machen zu müssen. All das ist aber in anderen Sprachen wie z. B. JAVA trotz aufwendigster Entwicklungsumgebungen immer noch der Fall. Mit SMALLTALK können Sie hingegen sofort loslegen.

Lernen durch
Ausprobieren

³ Man vergleiche dies mit dem Erklärungsnotstand, den eine Sprache wie JAVA bereits bei so einem simplen Beispiel mit sich bringt. Wer ECLIPSE und ähnlich mächtige IDEs zur JAVA-Programmierung benutzt, wird einwenden, dass dies „in JAVA auch geht“; tatsächlich geht es aber nicht „in JAVA“, sondern lediglich „in ECLIPSE“, also der IDE. Wer dieses Feature nicht kennt und es ausprobieren möchte: **File > New > Other > Java > Java Run/Debug > Scrapbook Page**. Es ist dies eines der vielen Erbstücke aus der Zeit, als die Entwicklerinnen von ECLIPSE noch an VISUALAGE saßen.



Ein weiterer wichtiger didaktischer Grund für die Wahl SMALLTALKS ist, dass es kein Typsystem hat. Bei den meisten anderen Sprachen müsste man mit Syntax und Semantik auch das Typsystem lernen und es fällt schwer, das Typsystem vom Rest der Sprache zu trennen. Objektorientierte Typsysteme unterscheiden sich aber zum Teil erheblich, ja sind sogar Gegenstand beinahe fanatisch geführter Auseinandersetzungen, obwohl sie mit den Grundgedanken der objektorientierten Programmierung zunächst nur wenig zu tun haben. Die Verwendung von SMALLTALK hingegen erlaubt, objektorientierte Programmierung unabhängig vom Typbegriff zu lehren und Typsysteme als das darzustellen, was sie sind: aufgepflanzte, semantische Regeln, die dazu dienen, logische Fehler in einem Programm zu finden, die aber zur eigentlichen Ausführung des Programms gar nicht notwendig sind.

keine Ablenkung durch Typsystem

Wenn also SMALLTALK Ihre erste objektorientierte Programmiersprache ist und Sie stattdessen lieber eine andere erlernt hätten, dann sollten Sie bedenken, dass es eine Vielzahl (die vermutlich auch zukünftig eher wachsen denn schrumpfen wird) verschiedener Programmiersprachen gibt, und dass der größte gemeinsame Teiler dieser Sprachen selbst keine existierende Programmiersprache ist. SMALLTALK ist jedoch, aufgrund seiner extremen Reduziertheit, recht dicht daran und beinahe alle objektorientierten Programmiersprachen wurden von SMALLTALK beeinflusst. Sie lernen also nicht für die *Beherrschung einer objektorientierten Programmiersprache*, sondern für ein *Verständnis aller*.

Latein der Objektorientierung

Programmierung mit SMALLTALK

Wenn Sie Ihr SMALLTALK-System nach der Installation das erste Mal starten, sehen Sie ein oder mehrere Fenster. Eines darunter ist das sogenannte Transcript, die „Konsole“ des SMALLTALK-Systems. Auf ihr werden bestimmte Meldungen vom System ausgegeben; sie entspricht im wesentlichen dem allgemeinen Ausgabestrom anderer Programmiersprachen, die dort z. B. über Print-Statements angesprochen werden. In JAVA entspricht das Transcript etwa `System.out`.

Transcript

Ausgehend vom Transcript können Sie weitere Fenster öffnen, und zwar entweder, indem Sie entsprechende Ausdrücke eingeben und auswerten, oder, indem Sie entsprechende Menübefehle aktivieren. (Wie das genau geht, erfahren Sie bei der Bearbeitung der Übungsaufgaben.) Zwei wichtige Arten von Fenstern sind sogenannte Workspaces und Klassenbrowser.

In einem Workspace können Sie Ausdrücke eingeben und auswerten lassen. Workspaces dienen in der Regel zum Herumexperimentieren. In einem Klassenbrowser können Sie sich die Klassen, aus denen das SMALLTALK-System, an dem Sie gerade arbeiten, besteht, anschauen und diese manipulieren. Einen solchen werden Sie vor allem gebrauchen, wenn Sie programmieren.

Workspaces und Klassenbrowser

Jeder Ausdruck, den Sie auswerten, und jede Klasse, die Sie ändern, bewirkt eine Veränderung des SMALLTALK-Systems. Sie wird, ohne dass Sie

Image und Change log



das merken, in einem Log file, gemeinhin „**Change log**“ genannt, mitgeschrieben. Wenn Sie SMALLTALK beenden, werden Sie gefragt, ob Sie das aktuelle **Image** speichern wollen. Das Image ist der Inhalt des (virtuellen) Speichers Ihres SMALLTALK-Systems, also all seine Objekte inklusive aller Dinge, die Sie sehen, also auch der Fenster und deren Inhalt. Wenn Sie das Image speichern, wird SMALLTALK sich beim nächsten Start genauso präsentieren, wie Sie es verlassen haben, inklusive aller Änderungen, inklusive aller Fenster und deren Inhalt. Es gibt also keinen Neustart — dazu müssten Sie das System schon neu installieren. SMALLTALK ist also wie ein reelles, physisches System, an dem Sie bauen (und das sich ja auch nicht jedes Mal in seine Ausgangsmaterialien zerlegt, wenn Sie ins Bett gehen).

Wenn Sie das Image nicht speichern, z. B. weil Sie eine fatale Änderung (solche gibt es und sind leicht möglich!) rückgängig machen wollen oder weil Ihnen der Strom ausgefallen ist, ist das kein Problem, denn Sie haben ja noch das Change log. Das Change log enthält wie gesagt alle Änderungen, die Sie gemacht haben, und zwar nicht in binärer Form, sondern als SMALLTALK-Ausdrücke, die Sie auswerten können. Wenn Sie das tun, wiederholen Sie damit Ihre gemachten, aber nicht im letzten Image gespeicherten Änderungen. Sie können den zuvor nicht gespeicherten Zustand Ihres Systems also ganz oder auch nur teilweise — ganz so, wie Sie es wünschen — wiederherstellen. Mit Hilfe des Change logs verlieren Sie in SMALLTALK nie auch nur eine einzige Zeile Code. Und das schon seit 1980.

Dieses wenige ist eigentlich alles, was Sie wissen müssen, um loszulegen. Welche Ausdrücke Sie sinnvollerweise eingeben und wie Sie programmieren, erfahren Sie in den Übungen zu diesem Vorkurs (im Rahmen der Abarbeitung der Einsendeaufgaben) sowie in den folgenden, ersten beiden Kurseinheiten.

Verfügbare SMALLTALK-Systeme

Aktuell gibt es mindestens drei ernstzunehmende SMALLTALK-Systeme, die Sie für diesen Kurs nutzen können:

- VISUALWORKS ist der direkte Nachfolger von SMALLTALK-80; es stellt die Weiterentwicklung des zunächst von den ursprünglichen Autoren selbst (unter dem Namen Parc Place Systems firmierend) vertriebenen originalen Systems dar. Gleichzeitig ist es unter den hier vorgestellten SMALLTALK-Systemen wohl das einzige, das auch für die kommerzielle Softwareentwicklung verwendet werden darf. Gleichwohl ist es für den akademischen Gebrauch frei erhältlich, und das auch noch für zahlreiche Plattformen. Wenn Sie sich jedoch von einem riesigen Funktionsumfang eher abgeschreckt als angezogen fühlen, ist VISUALWORKS eher nicht das Richtige für Sie.
- SQUEAK ist ein Ableger des originalen SMALLTALK-80-Systems, der von seinen beiden Vätern DAN INGALLS und ALAN KAY in eine etwas andere, strikt nicht-kommerzielle Richtung getrieben wurde, in die unter anderem die Unterrichtung von Kindern fällt. So enthält SQUEAK umfangreiche Multimediacibliotheken und damit Konstrukte, die man normalerweise nicht mit einer Programmiersprache assoziieren würde. Leider

Link



Link



beobachte ich unter Windows fortgesetzt Probleme mit den Ruhezuständen (Standby und Hibernate).

Link



- PHARO ist wiederum ein Ableger von SQUEAK, mit einem vergleichsweise aktiven Entwicklerteam. Es ist vermutlich das am Besten mit freien Lernmaterialien ausgestattete System und schon von daher eine Empfehlung wert.

Daneben gibt es noch einige andere, von denen ich zwei zumindest erwähnen möchte:

Link



Das heute immer noch im Netz herumgeisternde SMALLTALK EXPRESS ist eine in vielen Belangen vereinfachte und an die damalige Windows-PC-Hardware angepasste Version des ursprünglichen SMALLTALK-80. Da es mit einer vergleichsweise kleinen Klassenbibliothek und einer simplen Benutzungsschnittstelle auskommt, ist es für Anfängerinnen gut geeignet (und ist, wenn man es mit seinem 16-bit-Executable denn zum Laufen bekommt, meine persönliche Empfehlung).

Link



- AMBER ist der Versuch, SMALLTALK in einem Browser laufen zu lassen. Entsprechend der Philosophie SMALLTALKS (das ja ein selbstmodifizierendes System ist) können so Webanwendungen entwickelt werden. Erfahrungen damit habe ich keine.

Es ist Ihnen völlig freigestellt, welche der drei SMALLTALK-Systeme Sie im Verlauf dieses Kurses einsetzen. Für den Anfang empfehle ich SMALLTALK EXPRESS, aber wer sich mehr zutraut, die soll sich ruhig dahin wenden, wo sie hinwill: eher an kommerzieller Softwareentwicklung Interessierte zu VISUALWORKS, eher am Herumexperimentieren Interessierte zu SQUEAK. Mir persönlich ist SQUEAK am wenigsten zugänglich, aber das mag an meiner eigenen Historie liegen und an der allgemeinen Neigung, das am besten zu finden, was man am besten kennt. Sie sollen nur wissen, dass sich die drei Systeme in Details unterscheiden, zwar nicht in ihrer Syntax, aber in ihren Klassenbibliotheken und damit auch in ihrer Benutzung. Entsprechend fallen auch die Lösungen zu den Übungsaufgaben teilweise unterschiedlich aus.

Literatur zu SMALLTALK



Für alle mit einem Interesse an der Entwicklungsgeschichte von Programmiersprachen ist der Rückblick ALAN KAYS auf die Entstehung der Objektorientierung im allgemeinen und SMALLTALKS im speziellen unbedingt lesenswert. Sie werden nach der Lektüre einiges, das Sie heute als gegeben hinnehmen, mit ganz anderen Augen sehen.

Ein sehr gutes (wenn nicht das beste) Werk zur Einführung in SMALLTALK und die objektorientierte Programmierung ist das Buch „SMALLTALK-80“ von ADELE GOLDBERG und DAVID ROBISON. Es steht didaktisch mit Klassikern wie denen von KERNIGHAN & RITCHIE für C und WIRTH für PASCAL in einer Reihe. Da es sich bei dem Ihnen vorliegenden Text aber um einen Kurstext zur objektorientierten Programmierung und nicht zu SMALLTALK handelt und der Text deswegen auch Aspekte zu berücksichtigen hat, die in einer Einführung in SMALLTALK außen vor bleiben können, folgt dieser Kurstext nicht dem Aufbau des Buchs. Zudem habe ich mir

Link



erlaubt, allzu euphorische Kommentare der Autoren, insbesondere zu Modularität und Skalierbarkeit objektorientierter Programmierung, zu unterschlagen, da sie heute, wenn nicht überholt, so doch zumindest deutlich relativierungsbedürftig sind. Dennoch seien Leserinnen, die eine etwas langsamere Einführung in die Sprache bevorzugen, hiermit auf das Original verwiesen.

Eine wichtige Quelle für Bücher über SMALLTALK sind übrigens die Webseiten von STÉPHANE DUCASSE („Stef’s Free Online Smalltalk Books“). Dort gibt es auch den Artikel von ALAN KAY zum Herunterladen; allerdings ist die Qualität der Reproduktion ziemlich schlecht.

Link



Kurseinheit 1: Grundkonzepte der objektorientierten Programmierung

Ein laufendes objektorientiertes Programm muss man sich als eine Menge interagierender Objekte vorstellen. Damit die Objekte interagieren können, müssen sie verbunden sein; sie bilden dazu ein Geflecht, das neben Objekten aus Beziehungen zwischen diesen besteht. Das Geflecht verändert sich dynamisch infolge der Interaktion zwischen Objekten; es unterliegt aber gewissen, durch das Programm vorgegebenen statischen Strukturen.

Die Unterscheidung zwischen *Statik* und *Dynamik* ist eine klassische der Programmierung. Während Programme traditionell statische Gebilde sind, die auf Papier oder in einem Nur-lese-Speicher festgehalten werden können, ist ihre Ausführung immer etwas Dynamisches. Wenn aber Programme selbst als Daten aufgefasst werden, dann verwischt die Grenze zwischen Statik und Dynamik: Programme werden veränderlich. Insbesondere, wenn Programme sich selbst verändern können, ist die Unterscheidung zwischen Statik und Dynamik nur noch bedingt nützlich.

Alternativ zu Statik und Dynamik kann man auch zwischen *Struktur* und *Verhalten* unterscheiden, wobei mit Struktur das oben erwähnte Objektgeflecht und mit Verhalten die (Spezifikation der) Folge seiner Veränderungen gemeint ist. Diese Unterscheidung liegt der Gliederung des Rests dieser Kurseinheit zugrunde: von Objekten (Kapitel 1) und Beziehungen zwischen diesen (Kapitel 2) geht es über den Zustand als Bindeglied (Kapitel 3) zu den Elementen der Verhaltensbeschreibung (Kapitel 4).

Statik vs. Dynamik

Struktur vs. Verhalten

1 Objekte

In rein objektorientierten Programmiersprachen sind sämtliche Daten, die ein Programm verarbeiten kann, in Form von Objekten im Speicher abgelegt. Der Reiz dieses Merkmals der objektorientierten Programmierung ist, dass unser Weltbild, zumindest in weiten Teilen, auf einem ähnlichen Modell basiert: Die Welt besteht aus Objekten, die miteinander in Beziehung stehen. Dabei ist der Objektbegriff nicht auf das rein Materielle beschränkt: Nach allgemeinem Verständnis sind Personen ebenso Objekte wie Dokumente, Zahlen oder Zeichen.



Bei der Übertragung von realen (d. h., aus einer Anwendungsdomäne stammenden) Sachverhalten in ein objektorientiertes Programm ergibt sich das Problem, dass die Übertragung, aufgrund der Homogenität der Objektorientierung (alles ist ein Objekt), gewisse fundamentale Unterschiedlichkeiten der Kategorien unserer Begriffswelt ignoriert: Zahlen beispielsweise sind im Gegensatz zu Dingen Objekte ohne Identität, Zustand oder Lebensdauer (sie werden daher auch häufig als **Werte** bezeichnet); Mengen nicht weiter abgrenzbarer Elemente wie z. B. 1 Liter Wasser sind gar keine Objekte im eigentlichen Sinn (auch sie haben keine Identität) usw. Gleichwohl kommen sie alle in objektorientierten Programmen vor und werden dort — zumindest der reinen Lehre nach — durch Objekte repräsentiert. Der Ansatz, alles trotz evidenter ontologischer Unterschiede programmiersprachlich über einen Kamm zu scheren, führt hier und da zu gewissen Inkonsistenzen im ansonsten klaren, ja puristischen objektorientierten Weltbild, mit denen wir leben müssen, wenn wir objektorientiert programmieren wollen (vgl. dazu auch Kapitel 60 in Kurseinheit 6). Es ist dies der Preis des auch „Ockhams Rasiermesser“ genannten Spar- samkeitsprinzips, das auch für die objektorientierte Programmierung Leitlinie ist.

**verschiedene Arten
von Objekten**



WIKIPEDIA

1.1 Was ist ein Objekt?

Wie bereits erwähnt sind Objekte im Speicher abgelegte Daten. Dabei ist jedes Objekt an genau einer Stelle im Speicher abgelegt: Es wird damit durch seine Speicherstelle eindeutig identifiziert. Aufgrund dieser eindeutigen Identifizierbarkeit spricht man auch von der **Identität eines Objekts**; sie kann aus technischer Sicht mit der Speicherstelle, an der das Objekt abgelegt ist, gleichgesetzt werden. Da keine zwei Objekte an derselben Stelle abgelegt werden können, haben auch keine zwei Objekte dieselbe Identität.

Objekte sind grundsätzlich von *Werten* zu unterscheiden. Werte werden auch im Speicher abgelegt, haben aber keine Identität. Es folgt, dass derselbe Wert an verschiedenen Stellen im Speicher vorkommen kann. Viele objektorientierte Programmiersprachen (wie etwa JAVA oder C#) unterscheiden ganz offen zwischen Werten und Objekten; SMALLTALK tut dies nur hinter den Kulissen und folgt ansonsten seinem Motto „alles ist ein Objekt“.

Objekte vs. Werte

Die Menge des Speichers, den ein Objekt belegt, ist aus technischen Gründen konstant. Objekte können somit weder wachsen noch schrumpfen. Sollte dies trotzdem notwendig sein, bleibt nur, ein neues Objekt zu erzeugen, das an die Stelle (nicht die Speicherstelle!) des anderen tritt. Das neue Objekt hat jedoch eine andere Identität, so dass alle Stellen im Programm, die sich auf das alte Objekt beziehen, entsprechend angepasst werden müssen. Wie das geht, wird in Kurseinheit 2, Abschnitt 14.2 erläutert.

**Speicherbedarf von
Objekten**



1.2 Literale

Ein **Literal** (von lat. *littera*, der Buchstabe) ist eine in der Syntax der Programmiersprache ausgedrückte Repräsentation eines Objektes. Literale sind somit textuelle Spezifikationen von Objekten: Wenn der Compiler ein Literal übersetzt, erzeugt er daraus — bei der Übersetzung! — das entsprechende Objekt im Speicher. Dies steht im Gegensatz zu objekterzeugenden *Anweisungen* eines Programms, denn diese werden erst zur Laufzeit des Programms ausgeführt. Da wir uns mit der programmgesteuerten Erzeugung von Objekten aber erst in der nächsten Kurseinheit systematisch befassen, müssen wir hier zunächst mit Objekten mit literaler Repräsentation vorliebnehmen. Wohlgemerkt: Literale repräsentieren Objekte, es sind nicht selbst welche.

Die einfachsten Literale repräsentieren Zeichen (genauer: Zeichenobjekte). Um die literale Repräsentation eines Zeichens von anderen Vorkommen von Zeichen in einem Programm zu unterscheiden, wird ihnen in SMALLTALK ein \$-Zeichen vorangestellt. So bezeichnet das Literal

Zeichenliterale

2 **\$a**

das Zeichenobjekt „a“⁴. Dieses Objekt ist **atomar**, d. h., es ist nicht aus anderen Objekten zusammengesetzt. Zeichen sind in anderen Programmiersprachen — auch objektorientierten — übrigens typischerweise Werte.

Eine andere Art von Literalen, die atomare Objekte repräsentieren, sind die für Zahlen:

Zahlliterale

3 **1**

ist z. B. ein Literal, das das Objekt „1“ bezeichnet;

4 **-12.7**

ist ebenfalls ein Zahlliteral. Zahlliterale bezeichnen ebenfalls atomare (nicht zusammengesetzte) Objekte; sie sind in anderen Programmiersprachen typischerweise ebenfalls Werte (nicht jedoch sehr große Zahlen mit beliebiger Genauigkeit — die werden auch in anderen objektorientierten Sprachen durch Objekte repräsentiert).

Die in anderen Programmiersprachen vorzufindenden Literale (oder, je nach Sprache, *Schlüsselwörter*) **true**, **false** und **nil** (oder **null**), die genau wie Zeichen- und Zahlliterale atomare Objekte repräsentieren, sind in SMALLTALK nicht

Literale vs. (Pseudo-) Variablen

⁴ Ich verwende hier doppelte Anführungsstriche, um Objekte metasprachlich (also im Kurstext, nicht in einem Programm) zu benennen. Gemeint ist damit immer die Repräsentation des Objekts im Speicher. Um ein Literal, also die Repräsentation eines Objekts in einem Programm, zu benennen, setze ich es in diesem Kurstext in der Schriftart für (Programm-)Code.



Literale, sondern *Pseudovariablen* (s. Abschnitt 1.7). Der Grund dafür scheint eher pragmatischer Natur zu sein: SMALLTALK kennt keine Schlüsselwörter und indem man `true`, `false` und `nil` als (Pseudo-) Variablen auffasst, müssen sie vom Compiler syntaktisch nicht von *Variablen* (s. Abschnitt 1.5) unterschieden werden. So oder stehen sie für jeweils ein entsprechendes Objekt (die in anderen Sprachen wiederum Werte sind).

Wenn es atomare Objekte gibt, dann muss es auch **zusammengesetzte** geben. So können beispielsweise Zeichen zu **Zeichenketten**, den sogenannten **Strings**, zusammengesetzt werden, die ebenfalls Objekte sind. Ein String kann aber selbst wieder durch ein Literal bezeichnet werden; so steht in SMALLTALK

5 'Smalltalk'

für ein String-Objekt „Smalltalk“. Dieses Objekt ist selbst aus Objekten, nämlich den von den Zeichenliteralen `$S`, `$m`, `$a`, `$l`, `$1`, `$t`, `$a`, `$1` und `$k` repräsentierten Zeichenobjekten, zusammengesetzt. Was Zusammensetzung von Objekten heißt und wie sie funktioniert, darauf gehe ich in den Abschnitten 2.1 und 2.3 noch genauer ein.

Es repräsentieren also String-Literale zusammengesetzte Objekte. Daraus ergibt sich die Frage, ob zwei gleiche String-Literale dasselbe Objekt im Speicher repräsentieren. Dies ist nicht grundsätzlich so, wie wir noch sehen werden.

Um durch syntaktisch gleiche Zeichenketten stets dasselbe Objekt zu bezeichnen, bietet SMALLTALK sog. **Symbole** als weitere Art von Objekten mit literaler Repräsentation. So ist

6 #Smalltalk

die literale Repräsentation eines Objekts. Es bezeichnet bei jedem Vorkommen im Programm dasselbe Symbolobjekt „Smalltalk“ (nicht zu verwechseln mit dem obigen String-Objekt). Symbole dürfen, anders als Strings, nicht alle Zeichen enthalten (so z. B. keine Leerzeichen).

Da gleiche Symbolliterale immer dasselbe Objekt repräsentieren, ist die Erzeugung eines solchen Objekts durch den Compiler technisch aufwendiger als beispielsweise die anhand eines String-Literals. Der Compiler muss nämlich vor dem Erzeugen erst prüfen, ob das Literal schon einmal irgendwo vorkam. Ist das der Fall, erzeugt er kein neues Objekt, sondern verwendet stattdessen das bereits vorhandene. Das setzt natürlich eine entsprechende Verwaltung aller Symbolliterale und dazugehörigen Objekte durch den Compiler voraus.⁵ Wie man sich leicht vorstellen kann, wäre diese Vorgehensweise für die universell und in großer Anzahl verwendeten Strings sehr zeitaufwendig.

String-Literale

Symbolliterale

Verwaltung von Symbolliteralen

⁵ Die dafür verwendete Symboltabelle ist übrigens selbst ein SMALLTALK-Objekt mit Namen `SymbolTable`.



Gleichwohler versuchen manche SMALLTALK-Compiler, gleiche Literale, die zusammen kompiliert werden, auf dasselbe Objekt abzubilden. Das führt manchmal, durch das sog. *Aliasing* (s. Abschnitt 1.8), zu unerwarteten Ergebnissen bei der Verwendung dieser Literale.

Die letzte wichtige Kategorie von Literalen in SMALLTALK sind **Array-Literale**.

Array-Literale

rule. Die von ihnen repräsentierten Objekte sind genau wie Strings zusammengesetzt, bestehen aber im Gegensatz dazu nicht nur aus Zeichen, sondern aus einer Folge beliebiger, wieder durch Literale repräsentierter Objekte. Ein Array-Literal wird in SMALLTALK vom #-Zeichen und einer öffnenden Klammer angeführt, der durch Leerzeichen getrennte Literale folgen; es wird durch eine schließende Klammer abgeschlossen.

```
7 #(1 2 3)
```

ist ein solches Array-Literal,

```
8 #('Smalltalk' 'ist' 1.0 'Klasse')
```

ein anderes. Array-Literale können ineinander geschachtelt sein; das #-Zeichen entfällt jedoch bei allen inneren Arrays. In

```
9 #(($S $m $a $l $t $a $l $k) 'ist' 1.0 'Klasse')
```

beispielsweise ist das String-Literal 'Smalltalk' in Zeile 8 durch ein gleichbedeutendes Array-Literal, das aus Zeichenliteralen besteht, ersetzt.

Für Array-Literale gilt ansonsten das Gleiche wie für String-Literale: Dass zwei syntaktisch gleich sind heißt nicht, dass sie dasselbe Objekt erzeugen (oder, richtiger, dass aus ihnen nur ein Objekt erzeugt wird).

1.3 Änderbarkeit von Objekten

Während atomare Objekte grundsätzlich nicht änderbar sind (welchen Sinn hätte es beispielsweise, aus einer „1“ eine „2“ zu machen oder aus einem „a“ ein „b“?), so gilt das für zusammengesetzte zunächst nicht: Es ist leicht vorstellbar (und auch grundsätzlich sinnvoll), in einem Array-Objekt eine Komponente durch eine andere zu ersetzen. Die Frage ist allerdings, ob dies auch für Array-Objekte gilt, die aus Literalen erzeugt wurden: Soll es erlaubt sein, dass das zusammengesetzte Objekt, das aus dem Array-Literal #(1 2 3) hervorgegangen ist, durch ein Programm abgeändert wird, so dass es nicht mehr seiner (ursprünglichen) literalen Repräsentation im Programm entspricht? Dies ist Ansichtssache und wird zumindest für String- und Array-Literale von unterschiedlichen SMALLTALK-Dialektten unterschiedlich gehandhabt. Objekte, die aus Symbolliteralen hervorgegangen sind, sollten dagegen nie änderbar sein.



Grundsätzlich sind zusammengesetzte Objekte in SMALLTALK jedoch änderbar. Es ist dies Voraussetzung dafür, dass Objekte einen *Zustand* haben können (s. Kapitel 3), was wiederum die objektorientierte Programmierung zu einer Form der *imperativen Programmierung* macht. Durch die Zunahme *funktionaler* Einflüsse auf die objektorientierte Programmierung findet man jedoch auch zunehmend Sprachen, die unveränderliche Objekte anbieten (so z. B. SCALA).

1.4 Gleichheit und Identität von Objekten

Wie oben schon zur Unterscheidung von String- und Symbolliteralen angedeutet, wird durch das Vorkommen des gleichen Literals an mehreren Stellen eines Programms nicht notwendigerweise dasselbe, also identische, Objekt repräsentiert — es kann auch sein, dass die erzeugten Objekte nur gleich sind. Das wirft natürlich sofort die Frage auf, was der Unterschied zwischen Gleichheit und Identität bei Objekten ist, und wie überhaupt Objekte unterschieden werden können.

Die **Gleichheit von Objekten** ist Definitionssache und orientiert sich in der Regel an ihrem Erscheinungsbild oder ihrer Bedeutung. Gleichheit wird in SMALLTALK durch den Gleichheitsoperator `=` getestet. So liefern

Gleichheit von Objekten

```
10 $a = $a
11 1 = 1
12 1.0 = 1.0
13 'Smalltalk' = 'Smalltalk'
14 #Smalltalk = #Smalltalk
15 #(1 2 3) = #(1 2 3)
```

alle `true`. Aber auch

```
16 1 = 1.0
```

liefert `true`: Obwohl die beiden Zahlliterale verschieden sind (und für verschiedene, also zwei, Objekte stehen), bezeichnen sie doch (aus mathematischer Sicht) die gleiche Zahl, so dass man sie in SMALLTALK als gleich definiert hat.

Die **Identität zweier Objekte** (alternativ: die gleiche Identität zweier Objekte) wird in SMALLTALK durch `==` getestet. So liefern

Identität von Objekten

```
17 #Smalltalk == #Smalltalk
18 1 == 1
```

erwartungsgemäß `true`,

```
19 'Smalltalk' == 'Smalltalk'
```

kann hingegen zu `false` auswerten — zwei syntaktisch gleiche String-Literalen können also zwei Objekte mit verschiedener Identität repräsentieren. Dies ist zumindest dann sinnvoll,



wenn die durch die String-Literale erzeugten Objekte unabhängig voneinander änderbar sein sollen und deswegen tatsächlich zwei Objekte sein müssen. Übrigens: Phrasen wie „zwei identische Objekte“ sind strenggenommen Unsinn, denn es handelt sich bei vorliegender Identität definitionsgemäß nicht um zwei, sondern nur um ein Objekt. Die Frage nach der Identität von Objekten ist nur dann sinnvoll, wenn die Objekte durch *Namen* (oder *Variablen*) repräsentiert werden. Mehr dazu in Abschnitt 1.5.

Selbsttestaufgabe 1.1

Prüfen Sie in einem oder mehreren Ihnen zur Verfügung stehenden SMALLTALK-Systemen für verschiedene gleiche Literale, ob die repräsentierten Objekte identisch sind. Beschränken Sie sich dabei nicht nur auf die Beispiele der Zeilen 10–16. Was fällt Ihnen auf?

Während man sich unter der Identität einer Person oder eines Dokuments leicht etwas vorstellen kann, scheint der Begriff der Identität für manch andere Objekte merkwürdig. Was hat man sich beispielsweise unter der Identität der Zahl „1“ vorzustellen? Und wenn „1“ tatsächlich ein Objekt mit Identität ist, was macht dieses Objekt zur Eins? Oder ist die 1 vielleicht die Identität des Objekts „1“, sind also das Objekt und seine Identität dasselbe?

Ist es immer sinnvoll,
von Identität zu
sprechen?

Im Falle atomarer (also nicht zusammengesetzter) Objekte könnte man versucht sein, die Identität zweier Objekte mit der Gleichheit ihrer Erscheinungen gleichzusetzen: Es erscheint wenig sinnvoll, zwei immer gleiche Objekte mit unterschiedlicher Identität zu haben. So kann man sich beispielsweise fragen, warum man mehrere „1“ mit unterschiedlicher Identität in einem System haben sollte. Tatsächlich würde es wohl kaum auffallen, wenn zwei solche gleichen, aber sich dennoch aufgrund ihrer Identität unterscheidenden Objekte zu einem verschmelzen würden. Ganz anders ist das bei veränderlichen Objekten: Aufgrund ihrer Veränderlichkeit können sie sich auch nur vorübergehend gleichen, müssen aber selbst während dieser (vorübergehenden) Gleichheit voneinander zu unterscheiden sein, da sie sich hinterher wieder auseinanderentwickeln können und man dann nicht mehr wüsste, welches welches war. Da dies aber für unveränderliche Objekte nicht der Fall sein kann, ist es durchaus berechtigt, zu fragen, warum sie sich nur aufgrund ihrer Identität unterscheiden sollten.

Gleichsetzung von
Identität und
Erscheinung

Die Antwort ist vor allem technischer Natur. Wenn sich ein unveränderliches Objekt wie beispielsweise eine Zahl nicht aus einem Literal, sondern aus einer Operation (einer Rechenoperation) ergibt, dann müsste, für eine Zusammenlegung gleicher Objekte zu einem, immer erst überprüft werden, ob ein gleiches Objekt bereits angelegt wurde. Da dies Programme stark verlangsamen würde, nimmt man lieber in Kauf, mehrere gleiche, aber nicht identische Objekte zu haben. Aber warum sind dann gleiche Zahlen manchmal identisch, manchmal nicht? Die Antwort ist noch technischer: Sie hat etwas mit der Repräsentation von Objekten im Speicher zu tun und wird im nächsten Abschnitt gegeben. Und so werden in SMALLTALK bestimmte Objekte eben anders behandelt als der Rest: Ganze Zahlen (Integer) bis zu einer bestimmten Größe

warum auch
unveränderliche
Objekte verschiedene
Identitäten haben



und Zeichen sind aus technischen Gründen immer auch identisch, wenn sie gleich sind — für den Rest (mit Ausnahme der Symbole!) gilt das nicht. Konzeptuelle Gründe für die Sonderbehandlung gibt es nicht.

Zusammenfassend merken Sie sich am besten folgendes:

Quintessenz

„Das gleiche“ und „dasselbe“ sind auch in der objektorientierten Programmierung nicht das gleiche (und schon gar nicht dasselbe)!

Die Missachtung dieses Merksatzes ist eine der häufigsten Fehlerquellen der objektorientierten Programmierung. Besonders beim Vergleichen von Strings ist die Verwendung des Tests auf Identität anstelle des Tests auf Gleichheit ein häufiger logischer Programmierfehler. Deswegen noch einmal ganz deutlich:

Zwei Objekte können zwar gleich, aber nie dasselbe sein, oder es sind nicht zwei Objekte, sondern eins!

Verwenden Sie also grundsätzlich den Test auf Gleichheit (=), nicht auf Identität (==), es sei denn, Sie wollen prüfen, ob Sie es mit einem oder mit zwei Objekten zu tun haben.

1.5 Variablen

Weil Literale immer die gleichen Objekte repräsentieren, reichen sie zum Programmieren nicht aus. Was man vielmehr auch noch benötigt, sind **Namen**, die zu verschiedenen Zeitpunkten verschiedene Objekte bezeichnen können⁶, die sog. **Variablen**.

Genau wie ein Literal steht eine Variable in einem Programm für ein Objekt. Anders als bei Literalen wird aus einer Variable jedoch kein Objekt erzeugt: Sie ist lediglich ein Name für ein bereits existierendes Objekt. Dazu kommt, dass eine Variable zu unterschiedlichen Zeitpunkten für unterschiedliche Objekte stehen kann (deswegen der Name „Variable“!). Es können zudem Variablen mit unterschiedlichen Namen für dasselbe Objekt stehen, das damit gewissermaßen verschiedene Namen hat (die sog. *Aliase*; s. Abschnitt 1.8). Wir werden daher im folgenden davon sprechen, dass Variablen Objekte benennen oder bezeichnen.

Variable vs. Literal

⁶ Achtung: In der *funktionalen Programmierung*, in der der Begriff des Namens gebräuchlicher ist als in der objektorientierten, steht ein Name immer für dasselbe Objekt.



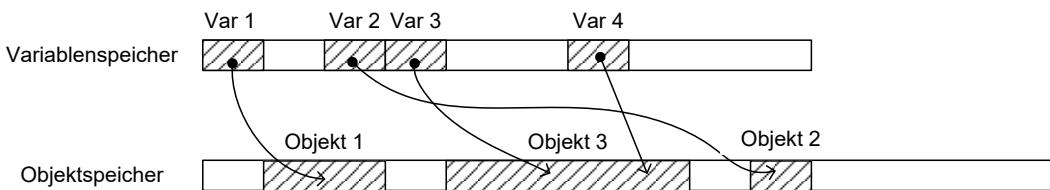
1.5.1 Inhalt

Das bezeichnete Objekt wird manchmal auch „Wert“ oder „Inhalt“ der Variable genannt (und die Variable selbst *Platzhalter* des Objekts). Besonders die Verwendung von „Inhalt“ ist aber gefährlich, da sie nahelegt, ein Objekt könne zu einem Zeitpunkt nur von genau einer Variable bezeichnet werden, so wie ein Gegenstand zu einer Zeit immer nur Inhalt eines Behälters sein kann. Tatsächlich können aber mehrere Variablen gleichzeitig ein und dasselbe Objekt bezeichnen — die Variablen haben nämlich nur **Verweise** (auch **Referenzen** oder **Pointer** genannt) auf Objekte (genauer: auf die Speicherstellen, an denen die Objekte abgelegt sind; s. o.) zum Inhalt. Man spricht deswegen auch von einer **Verweis-** oder **Referenzsemantik** von Variablen, im Gegensatz zur **Wertsemantik**, bei der das bezeichnete Objekt tatsächlich auch Inhalt der Variable ist.

**Wert- und
Verweissemantik von
Variablen**

Aus technischer Sicht entspricht einer Variable eine Stelle im Speicher. Allerdings steht an dieser Stelle bei Variablen mit Verweissemantik nicht das Objekt, das sie bezeichnen, sondern lediglich ein Verweis auf die Speicherstelle, an der das Objekt gespeichert ist. Es handelt sich also bei Variablen mit Verweissemantik aus technischer Sicht um **Pointervariablen**, wie man sie auch aus nicht objektorientierten Programmiersprachen wie PASCAL oder C kennt.

**Variableninhalt auf
Speicherebene**



Verweis- und Wertsemantik von Variablen unterscheiden sich fundamental: Unter Wertsemantik können, solange jedes Objekt seine eigene Identität hat, zwei Variablen niemals dasselbe Objekt bezeichnen. Dies wird aber nur den wenigsten Programmierproblemen gerecht. Da zudem die Verweissemantik einen wesentlich speicher- und recheneffizienteren Umgang mit Objekten erlaubt und da unterschiedliche Objekte wie oben beschrieben unterschiedlich viel Speicherplatz belegen, so dass man im Vorfeld nicht immer weiß, wie viel davon man für eine Variable vorsehen muss, ist sie in der objektorientierten Programmierung vorherrschend. In manchen Sprachen, die neben Objekten auch Werte kennen, haben Variablen, die Objekte aufnehmen, stets Verweissemantik und Variablen, die Werte aufnehmen, stets Wertsemantik (z. B. JAVA); andere objektorientierte Sprachen erlauben der Programmiererin, für jede Variable getrennt festzulegen, ob sie Wert- oder Verweissemantik haben soll (so z. B. C++ und EIFFEL).

**Bedeutung der
Unterscheidung von
Verweis- und
Wertsemantik**

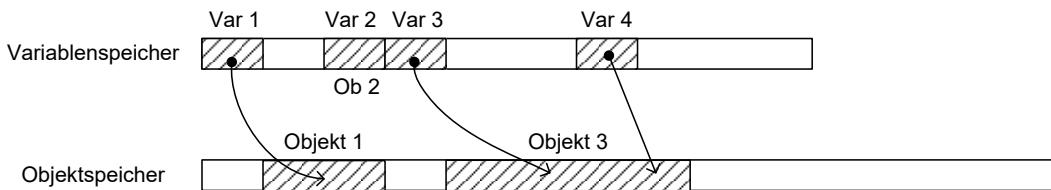
Nun ist besonders für unveränderliche Objekte, deren interne Repräsentation klein ist (die also wenig Speicherplatz belegt), die Forderung nach der Speicherung eines Objektes an genau einem Ort und Speicherung von

**wechselnde
Semantik von
Variablen in
SMALLTALK**

Verweisen in Variablen (also die Speicherung in Variablen mit Verweissemantik) ineffizient. Welchen Sinn hätte es beispielsweise, allen Zeichen eine Identität zu geben, an der mit der jeweiligen Identität verbundenen Stelle im Speicher die internen Repräsentationen zu hinterlegen und dann in Variablen die Speicherstelle (Identität) zu speichern, wenn der Verweis mehr Speicher belegt als das Zeichenobjekt, auf das verwiesen wird? Das Gleiche gilt auch für Zahlen bis zu einer gewissen Größe.

In den meisten SMALLTALK -Implementationen hat man dieses Problem so gelöst, dass Variablen, die Zeichen, kleine Zahlen und die Booleschen Werte `true` und `false` bezeichnen, Wertsemantik haben. Die Objekte können damit aber tatsächlich an mehreren Stellen im Speicher gespeichert werden, was einen Widerspruch zur reinen Lehre darstellt. Zwar geht damit der Begriff der Identität für diese Objekte verloren, aber für die Programmiererin ist die damit verbundene mehrfache Existenz identischer Objekte im Speicher insofern ohne größere Bedeutung, als hier Gleichheit problemlos an die Stelle der Identität treten kann. Der Preis für diese Flexibilität ist allerdings, dass man den Variablen nicht mehr fix Wert- oder Verweissemantik zuordnen kann — diese hängt vielmehr jeweils von der Art der Objekte ab, die sie gerade bezeichnen. In diesem Fall würde man Wert- bzw. Verweissemantik eher als eine Eigenschaft des Objekts denn der Variable ansehen; das ist jedoch ziemlich SMALLTALK-spezifisch.

Wertsemantik bei kleinen Objekten



1.5.2 Sichtbarkeit

Eine Variable bezeichnet also ein Objekt. Wer auf eine Variable zugreifen kann, kann damit automatisch auch auf das Objekt zugreifen, das die Variable bezeichnet. Tatsächlich sind alle Objekte, für die es keine eindeutige literale Repräsentation (wie sie Zeichen, manche Zahlen und Symbole haben) gibt, nach ihrer Erzeugung nur noch über Variablen zugreifbar. Die einzige Ausnahme bilden hier die sog. *konstanten Methoden*, die jedoch erst in Abschnitt 4.3.6 behandelt werden.

Nun ist es nicht sinnvoll, dass in einem Programm alle Variablen (und damit auch alle Objekte) von überall her zugreifbar sind. Um den Zugriff auf Variablen einzuschränken, gibt es den Begriff der *Sichtbarkeit* und *Regeln für die Sichtbarkeit von Variablen*. Kurzgefasst ist die Sichtbarkeit einer Variable gleichbedeutend damit, dass man ihren Namen verwenden kann (und damit auch Zugriff auf das von diesem Namen bezeichnete Objekt hat). Dabei bezieht sich die Sichtbarkeit immer auf einen Abschnitt von Programmcode: Wenn eine Variable in einem Abschnitt sichtbar ist, dann entspricht jedes Vorkommen des Variablennamens in diesem Abschnitt einer ihrer Verwendungen.



Die einzelnen Programmiersprachen unterscheiden sich zum Teil deutlich in der Definition ihrer Sichtbarkeitsregeln. Häufig wird jedoch zwischen sog. **globalen** und **lokalen Variablen** unterschieden. Dabei sind beide Begriffe relativ zu verstehen: lokale Variablen sind in ihrer Sichtbarkeit auf den Programmabschnitt beschränkt, um den es gerade geht (sowie ggf. auf dessen Unterabschnitte), globale Variablen sind auch außerhalb davon (insbesondere in Überabschnitten) sichtbar. Variablen, die überall sichtbar sind, sind also immer (relativ zu jedem Programmabschnitt) global. Wenn eine Variable außerhalb eines bestimmten Programmabschnitts, aber nicht überall sichtbar ist, sagt man auch, sie sei global zu dem Programmabschnitt; sie ist dann lokal zu einem übergeordneten (umschließenden) Programmabschnitt. Lokale Variablen überdecken übrigens immer globale Variablen gleichen Namens; man spricht dann auch von *Hiding*.

In SMALLTALK müssen globale Variablen mit einem Großbuchstaben beginnen. **Smalltalk** und **Transcript** sind zwei prominente Beispiele für globale Variablen. Lokale Variablen beginnen hingegen mit einem Kleinbuchstaben und sind auf den Sichtbarkeitsbereich eines Objekts (oder auch nur Teilen davon) beschränkt. Für die genaue Angabe der *Sichtbarkeitsregeln* SMALLTALKS fehlt uns noch einiges; wir werden daher erst in den folgenden Abschnitten darauf eingehen; wir können aber schon hier schlussfolgern, dass in SMALLTALK der Unterschied zwischen lokal und global nicht relativ ist (es also nur zwei verschiedene Programmabschnitte gibt).

**Benennungs-
konvention in
SMALLTALK**

Selbsttestaufgabe 1.2

Versuchen Sie, durch Experimentieren herauszufinden, was in der Variable **Smalltalk** zu finden ist.

1.6 Zuweisung

Damit eine Variable ein Objekt bezeichnet, muss ihr dieses durch eine sog. **Zuweisung**, in anderen Kontexten auch **Wertzuweisung** genannt, zugeordnet werden. Ursprünglich wurde als Zuweisungsoperator das Symbol „←“ gewählt; wegen der mangelnden Verfügbarkeit auf Tastaturen wurde es jedoch in den meisten SMALLTALK-Implementierungen durch das aus ALGOL und PASCAL bekannte := (englisch als „becomes“ gelesen) ersetzt.⁷ Die Variable **lieblingszahl** bezeichnet also in Folge der Zuweisung

²⁰ **lieblingszahl := 2**

ein Objekt „2“ (in der Zuweisung repräsentiert durch das Literal 2). Nach einer Zuweisung

⁷ Dass in C und allen davon abgeleiteten Sprachen (sowie in BASIC) das einfache Gleichheitszeichen = für die Zuweisung steht, darf als eine der Tragödien in der Geschichte der Programmiersprachen angesehen werden. Ich möchte nicht wissen, wie viele fatale Fehler auf die dadurch provozierte Verwechslung von Test auf Gleichheit und Zuweisung zurückzuführen sind.



21 `x := y`

bezeichnen `x` und `y` das gleiche Objekt (genau welches ist hier nicht ersichtlich); ob sie auch dasselbe bezeichnen, hängt von der Semantik der Variablen ab. Man beachte, dass in SMALLTALK (anders als in typisierten Sprachen) aus Sicht des Compilers nichts dagegenspricht, der Variable `x` erst eine Zahl und dann einen String zuzuweisen. Auch Array-Literale können jeder beliebigen Variable zugewiesen werden.

keine Einschränkungen bei der Zuweisung

Man beachte weiterhin, dass die Zuweisung (anders als der Test auf Gleichheit = oder Identität ==) nicht kommutativ ist: `x := y` hat nur dann

zwei Seiten einer Zuweisung

dieselbe Bedeutung wie `y := x`, wenn `x` und `y` schon vor der jeweiligen Zuweisung denselben Wert hatten. Zur besseren sprachlichen Unterscheidung der Seite, der zugewiesen wird, und der, die zugewiesen wird, spricht man häufig von der *linken und der rechten Seite einer Zuweisung*.

Nach den drei Zuweisungen

Beispiel

22 `x := 5`
23 `y := 3`
24 `x := y`

bezeichnen `x` und `y` beide die „3“. Wäre die letzte Zuweisung hingegen `y := x` gewesen, bezeichneten `x` und `y` beide „5“.

Die Zuweisung ist ein elementares Konstrukt der objektorientierten Programmierung sowie der Programmierung überhaupt. Nur die wenigsten Sprachen kommen ohne sie aus. Neben der expliziten Zuweisung durch den Zuweisungsoperator kommt auch eine *implizite* (bei *Methodenaufrufen*) vor; diese wird jedoch erst in Abschnitt 4.3.2 behandelt.

Der oben geschilderte Unterschied zwischen Wert- und Verweissemantik von Variablen hat für die Zuweisung erhebliche Konsequenzen: Bei einer Zuweisung unter Wertsemantik muss, da die Variable das Objekt zum *Inhalt* hat (also in der Variable gespeichert ist) und *ein* Objekt nicht *in zwei* Variablen gespeichert sein kann, eine Kopie angefertigt werden. Das hat zur Folge, dass die beiden Variablen `x` und `y` nach der Zuweisung aus Zeile 21 nicht dasselbe (also identische) Objekt bezeichnen (was ja unter Wertsemantik, wie oben bereits gesagt, auch gar nicht geht), so dass z. B. Änderungen am in `x` gespeicherten Objekt das in `y` gespeicherte Objekt nicht betreffen. Bei einer Zuweisung unter Verweissemantik wird jedoch nur der Verweis der rechten Seite kopiert und in der Variablen auf der linken gespeichert. Wenn die Variablen auf der linken und der rechten Seite unterschiedliche Semantiken haben, dann liegt entweder eine *unzulässige Zuweisung* (s. Kapitel 18) vor oder es muss, je nach Art der Variable auf der linken Seite, eine Kopie eines Objektes oder ein Verweis auf ein Objekt angefertigt werden (s. dazu auch Abschnitt 52.5.2 in Kurseinheit 5).

Wert- und Verweissemantik bei der Zuweisung



Selbsttestaufgabe 1.3

Finden Sie (durch Experimentieren) heraus, welche Objekte Ihres SMALLTALK-Systems per Wertsemantik gespeichert werden. Nutzen Sie dabei aus, das SMALLTALK vor der Erzeugung eines Objekts mit Ausnahme von Symbolen nicht prüft, ob das Objekt schon vorhanden ist.

Hinweis: Verwenden Sie den Identitätstest (`==`).

1.7 Pseudovariablen

Während es für Variablen charakteristisch ist, dass sich ihr Wert ändern kann, so sieht SMALLTALK dennoch einige vor, für die das nicht der Fall ist. Hier sind vor allem die Variablen mit Namen `true`, `false` und `nil` zu nennen, die auf Objekte entsprechender Bedeutung verweisen.⁸ Für diese Variablen ist die Zuweisung nicht zulässig.

Eine ganze Reihe weiterer Variablen kann zwar ihren Wert ändern (also zu unterschiedlichen Zeiten auf verschiedene Objekte verweisen), jedoch erhalten sie ihren Wert vom System; auch diesen kann durch den Zuweisungsoperator `:=` kein Wert zugewiesen werden. Dies sind z. B. die Variablen mit Namen `self` und `super` sowie alle *formalen Parameter* von Methoden (s. Abschnitt 4.3). Nicht zuletzt sind auch alle Klassennamen (s. Kurseinheit 2) Variablen, denen man als Programmiererin nichts explizit zuweisen kann. All diese Variablen werden in SMALLTALK einheitlich **Pseudovariablen** genannt.

1.8 Aliasing

Wenn Variablen keine Objekte enthalten, sondern lediglich auf sie verweisen, wenn sie also Verweissemantik haben, ist es möglich, dass mehrere Variablen gleichzeitig dasselbe Objekt benennen. Das nennt man **Aliasing**. Das Aliasing ist eines der wichtigsten Phänomene der objektorientierten Programmierung; zugleich ist es leider nur wenig als solches bekannt. Versuchen Sie trotzdem, es sich stets bewusst zu machen — es wird Sie vor manch böser Überraschung bewahren.

Aliase, also weitere Namen für ein bereits benanntes Objekt, entstehen immer bei der Zuweisung. Dazu ist es notwendig, dass die Variable auf der linken Seite Verweissemantik hat. Da in SMALLTALK die Semantik von Variablen nicht mit der Variablen Deklaration (s. Kapitel 19) festgelegt wird, sondern von der Art eines Objekts abhängt, ist nicht immer klar, bei welcher Zuweisung ein Alias entsteht. Dabei kann beides, die fälschliche Annahme von Verweissemantik bei tatsächlicher Wertsemantik und die

Entstehung von
Aliasing

⁸ Die Pseudovariablen `true`, `false` und `nil` benennen spezielle, unveränderliche Objekte, auf deren Bedeutung wir noch ausführlich eingehen werden. Bis dahin kann die Leserin davon ausgehen, dass `true` und `false` für die Booleschen Werte „wahr“ und „falsch“ stehen und `nil` für ein spezielles Objekt, das meistens „kein Objekt“ repräsentieren soll.



fälschliche Annahme von Wertsemantik bei tatsächlicher Verweissemantik, zu erheblichen (und schwer zu findenden) Programmierfehlern führen.

Nach den beiden Zuweisungen

Beispiel

```
25 x := #Smalltalk  
26 y := #Smalltalk
```

hat das eine Objekt, das der Compiler für `#Smalltalk` erzeugt, zwei Namen, nämlich `x` und `y`.

Das Aliasing ist zunächst erwünscht: Da jedes Objekt nur einmal im Speicher hinterlegt werden muss, ermöglicht es die extrem effiziente Informationsverarbeitung (es ist weder ein Kopieren notwendig, wenn ein Objekt weitergereicht werden soll, noch müssen die Änderungen an den verschiedenen Kopien zusammengeführt werden, die notwendig sind, wenn die Kopien immer noch dasselbe logische Objekt bezeichnen sollen). Doch diese Effizienz hat ihren Preis.

Effizienz durch
Aliasing

Dass die Veränderung des durch eine Variable bezeichneten Objekts zugleich die Veränderung der durch all seine Aliase bezeichneten Objekte (die ja alle dieselben sind) bewirkt, kann nämlich unerwünscht, ja ein Programmierfehler sein. So könnte man beispielsweise bei den beiden Zuweisungen

mögliche Probleme
durch Aliasing

```
27 petersNachname := 'Müller'  
28 paulasNachname := petersNachname
```

lediglich bezwecken wollen, dass Peter und Paula *zunächst* gleich heißen, z. B. weil sie Geschwister sind. Bei einer späteren Promotion Paulas fügt sie die Zeichen `$D`, `$r` und `$.` in den ihren Nachnamen repräsentierenden String ein, ändert also das entsprechende Objekt. Man hat nun sicher nicht beabsichtigt, dass das auch `petersNachname` betrifft, aber wenn die Änderung an einer weit entfernten Stelle im Programm erfolgt, ist die Identität der von `petersNachname` und `paulasNachname` benannten Objekte nicht mehr offensichtlich. Tatsächlich hat man es dann mit einem recht subtilen und schwer zu findenden Programmierfehler zu tun. Deswegen (und aufgrund etwas überzeugenderer Beispiele, die zu verwenden aber noch mehr Vorbereitung bedarf) sind in einigen SMALLTALK-Systemen alle auf Basis literaler Repräsentationen erzeugten Objekte als unveränderlich markiert (wenn Sie es nicht schon, wie beispielsweise Zahlen, von Haus aus sind), so dass Programmierfehler dieser Art vermieden werden. Sollte wie im obigen Beispiel eine Zuweisung mit Wertsemantik benötigt werden, so schreibt man statt Zeile 28 in SMALLTALK einfach

```
29 paulasNachname := petersNachname copy
```

Dabei sorgt das hintangestellte `copy` dafür, dass von dem Objekt, das durch `petersNachname` bezeichnet wird, eine Kopie angefertigt wird, also ein neues Objekt, das dem alten gleicht (mehr zur Syntax und dazu, wofür `copy` steht, folgt unten). Nicht nötig wird das Kopieren, wenn ich die Änderung durch die Zuweisung eines neuen Objekts bewerkstellige, wie das beispielsweise in



```
30 paulasNachname := 'Dr. Müller'
```

oder gar

```
31 paulasNachname := 'Dr. ' , paulasNachname
```

der Fall ist (wobei das Komma hier für die *String-Konkatenation* steht).

Fehler dieser Art sind häufig die Folge dessen, dass sich eine Programmiererin der aliasbildenden Wirkung der Zuweisung nicht bewusst war. Das ist insbesondere bei den Programmierinnen der Fall, die nicht mit der objektorientierten Programmierung großgeworden sind, die insbesondere bei einer Zuweisung $y := x$ das Kopieren des Inhalts der Variablen auf der rechten Seite (x) vermuten. Tatsächlich muss man in anderen Sprachen (wie beispielsweise PASCAL oder C) eine Variable explizit als *Pointervariable* deklarieren, um einen Alias bilden zu können. In SMALLTALK, genau wie in JAVA und C#, ist Aliasing jedoch der Regelfall und Kopie die Ausnahme. Wer das nicht verinnerlicht hat, schreibt höchstens zufällig korrekte Programme.

1.9 Lebenslauf von Objekten

In SMALLTALK beginnt der Lebenslauf eines Objekts mit seiner Erzeugung und endet mit seiner Entsorgung durch eine Speicherbereinigung, die sog. **Garbage collection**. Garbage collection ist ein Mechanismus, der Objekte aus dem Speicher entfernt, wenn diese nicht mehr zugreifbar sind. Da in SMALLTALK auf Objekte nach ihrer Erzeugung ausschließlich über Variablen (Namen) zugegriffen werden kann, kann ein solches Objekt genau dann entfernt werden, wenn keine Variable mehr auf es verweist. Es *kann* entfernt werden, muss aber nicht; aus Sicht der Programmiererin ist es ausreichend, dass das Objekt nicht mehr bekannt/benannt ist — es kann somit nicht mehr aufgefunden und durch eine Zuweisung einer Variable zugewiesen werden. Bei der Implementierung von Garbage-collection-Algorithmen besteht denn auch erhebliche Freiheit.

Wenn Peter und Michaela heiraten, dann schlägt sich dies u. a. in der Zuweisung

Beispiel

```
32 petersNachname := michaelasNachname
```

nieder. Wenn 'Müller' keine Aliase (wie beispielsweise **paulasNachname**) hatte, kann es nach der Zuweisung aus dem Speicher entfernt werden — es wäre selbst bei Bedarf nicht mehr auffindbar.

Von der automatischen Speicherbereinigung ausgenommen sind bestimmte Objekte mit eindeutiger literaler Repräsentation (wie z. B. kleine Zahlen, Zeichen und Symbole). Im Falle von Zahlen und Zeichen liegt das jedoch weniger an der Natur dieser Objekte selbst als vielmehr an der Tatsache, dass diese in der Regel nicht als Objekte im Speicher angelegt werden (so dass Variablen darauf verweisen könnten),

**Sonderbehandlung
bestimmter Objekte**

sondern dass sie selbst, als Werte (und anstelle von Zeigern), in Variablen gespeichert werden (Abschnitt 1.5.1). Sie werden „entfernt“, indem einer Variable ein neuer Wert zugewiesen wird. Symbole werden schon deswegen nicht aus dem Speicher entfernt, weil sie in einer Symboltabelle abgelegt (und somit immer mindestens einmal referenziert; s. Fußnote 5) werden.

Der Mut zur Verabschiedung von der expliziten Speicherfreigabe war eine der wichtigsten Entscheidungen beim Entwurf SMALLTALKs. Man hat einfach anerkannt, dass die genaue Buchführung darüber, auf welche Objekte noch zugegriffen wird, zu schwierig ist, um die Verantwortung dafür Anwendungsprogrammiererinnen zu überlassen. Wer das Problem nicht unmittelbar einsichtig ist, die halte sich vor Augen, dass

warum automatische Speicherbereinigung richtig und wichtig ist

- der Ort der Erzeugung eines Objektes und seine erste Zuweisung zu einer Variable im Programm möglicherweise weit entfernt sind von der Stelle, an der dieser Variable ein anderes Objekt zugewiesen wird, dass
- es möglicherweise viele solcher Stellen gibt, von denen mal die eine, mal die andere zuerst erreicht wird, und dass
- in der Zwischenzeit beliebig viele Aliase auf das Objekt angelegt worden sein können, die alle mitberücksichtigt werden müssen, um zu entscheiden, ob das Objekt noch in Verwendung ist.

Eine vorzeitige Entfernung aus dem Speicher hingegen führt dazu, dass Variablen ins Nichts zeigen (eine häufige Quelle von Programmabstürzen) oder dass, bei einer Wiederverwendung des Speichers, die Variable plötzlich auf ein anderes Objekt verweist, das ihr aber nie explizit zugewiesen wurde — ein quasi zufälliges Programmverhalten, das mit hoher Wahrscheinlichkeit zu schweren Programmfehlern führen würde. Ein anderes Beispiel entsteht, wenn in einer Verzweigung eines Programms entweder ein neues oder ein bereits vorhandenes Objekt einer Variable zugewiesen wird. Woher weiß man bei der weiteren Benutzung dieser Variable, ob das Objekt schon vorher existierte und vielleicht schon andere Variablen auf es verweisen, oder ob es gerade erst neu erzeugt wurde und damit noch unbunutzt ist? Wer ist für die Entsorgung des Objekts verantwortlich? All diese Betrachtungen kann man sich in Gegenwart der Garbage collection ersparen.

Im objektorientierten Jargon spricht man übrigens häufig auch vom **Lebenszyklus** („life cycle“) eines Objekts. Genaugenommen ist dies aber irreführend, denn das Wort „Zyklus“ verspricht, dass das Leben nach seinem Ende wieder neu beginnt. Gerade dies ist aber, wie eben erläutert, nicht der Fall: Objekte werden nicht recycelt, sondern höchstens der Speicherplatz, den sie belegen.

„Lebenszyklus“ ist irreführend



1.10 Zusammenfassung

Objekte können also über textuelle Repräsentationen, die sog. *Literale*, in ein Programm eingeführt und *Variablen* zugewiesen werden. Dabei speichern die Variablen die Objekte in aller Regel nicht, sondern benennen sie lediglich. Technisch gesehen enthalten die Variablen dann *Verweise* (Referenzen, Zeiger, Pointer) auf die Objekte, die selbst durch Stellen im Speicher repräsentiert werden (sog. *Referenzsemantik*). Wenn mehrere Variablen dasselbe Objekt bezeichnen, spricht man von einem *Aliasing*. Aliasing erlaubt über die gemeinsame Nutzung von Objekten eine außerordentlich speicher- und recheneffiziente Informationsverarbeitung. Gleichzeitig stellt es aber eine der größten Fehlerquellen der objektorientierten Programmierung dar, da im allgemeinen nicht bekannt ist, ob und wie viele Aliase auf ein Objekt existieren. Ein besonderes Problem liegt vor, wenn sich die Programmiererin nicht bewusst ist, dass sie (nur) mit Aliasen hantiert, und sich dann wundert, wenn sich an entfernten Orten plötzlich die (vermeintlichen, denn eigentlich sind es ja gar keine) Inhalte von Variablen ändern.

2 Beziehungen

Kein Objekt ist eine Insel. Ganz im Gegenteil: Damit Objekte eine Bedeutung haben, müssen sie mit anderen in Beziehung stehen. Das Objekt „1“ beispielsweise ist ohne Bedeutung, solange es nicht eine bestimmte Eigenschaft eines anderen Objekts beschreibt, wie z. B. die Hausnummer eines Hauses oder die Anzahl der Elemente eines Arrays. Tatsächlich werden die meisten Objekte eines Systems erst durch ihre Beziehungen zu anderen zu etwas Nützlichem. Ein Objekt, das beispielsweise eine Person repräsentiert, macht den String „Hans Mustermann“ zum Namen der Person, sobald er mit der Person in entsprechender Beziehung steht; umgekehrt wird für die Benutzerin des Systems die Person erst über den Namen identifizierbar.

Tatsächlich wird, wie bereits eingangs dieser Kurseinheit erwähnt, in der objektorientierten Programmierung sämtliche Information als ein *Geflecht von Objekten* dargestellt. Dieses Geflecht kann

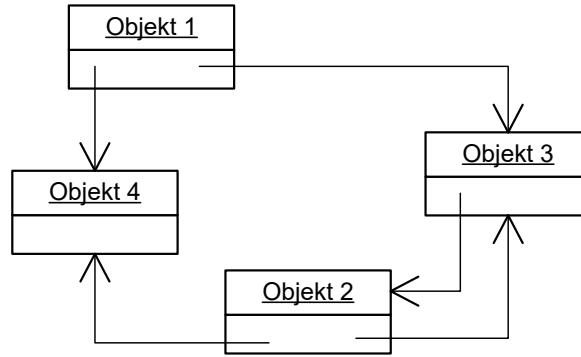
**Information als
Geflecht von
Objekten**

1. navigiert werden, um von einem Datum (Stück Information) zu einem anderen zu kommen, oder auch
2. manipuliert werden, um die repräsentierte Information zu verändern.

Das Datenmodell der objektorientierten Programmierung ähnelt damit stark dem *Netzwerkmodell*, das ebenfalls ein navigierendes ist, das vor einigen Jahrzehnten einmal die Grundlage großer Datenbankmanagementsysteme war, das aber schnell vom *relationalen Datenmodell* verdrängt wurde und das erst heute, im Zuge der Einführung von objektorientierten Datenbanken, wieder an (theoretischer) Bedeutung gewinnt.



Kurs



Richtung von Beziehungen

In der objektorientierten Programmierung werden Beziehungen zwischen Objekten über Verweise hergestellt, durch deren Verfolgung man von einem Objekt zum nächsten gelangen, eben „navigieren“ kann. Das Besondere dieser Verknüpfung ist, dass sie stets gerichtet ist: Dass man von einem Objekt zum nächsten navigieren kann, heißt nicht, dass man auch wieder zurückkommt. Dazu ist dann ein Zeiger in Gegenrichtung nötig.

Nun enthalten ja Variablen ebenfalls Verweise. Wer also Zugriff auf die Variable hat, hat damit auch Zugriff auf das referenzierte Objekt — und ist somit mit dem Objekt verknüpft. Was noch fehlt, ist, dass Variablen Objekten so zugeordnet werden, dass nur noch die Objekte Zugriff darauf haben, und schon kann man auf einfache Weise Beziehungen ausdrücken.

2.1 Instanzvariablen

Jedem Objekt kann eine Menge von *lokalen Variablen* zugeordnet werden. Aus Gründen, auf die wir noch zu sprechen kommen, heißen diese Variablen **Instanzvariablen**; sie werden aber manchmal auch *Felder* oder *Attribute* (zu Attributen s. Abschnitt 2.4) genannt. Die Instanzvariablen eines Objekts sind in gewisser Weise in seinem Besitz: Sie sind für andere Objekte nicht sichtbar und damit auch nicht zugreifbar. Die Sichtbarkeit ist also auf das jeweils besitzende Objekt eingeschränkt.⁹ Außerdem ist die Existenz dieser Variablen an die Existenz (oder Lebensdauer; s. Abschnitt 1.9) des besitzenden Objekts gebunden.

Instanzvariablen bestimmen den Aufbau, oder die **Struktur**, zusammengesetzter Objekte (die manchmal deswegen auch *strukturierte Objekte* genannt werden) — atomare Objekte haben keine Instanzvariablen. Jede Instanzvariable eines Objekts belegt dabei einen Teil des Speichers aus seiner Repräsentation

Instanzvariablen und die Zusammensetzung von Objekten

⁹ Während das in SMALLTALK Gesetz ist, weichen andere Sprachen (wie z. B. JAVA) diese Regel auf, indem sie öffentlich zugängliche Instanzvariablen erlauben und selbst für private nicht verhindern, dass ein anderes Objekt der gleichen Klasse darauf zugreift. Etwas anderes ist es jedoch auch in SMALLTALK mit den Objekten, auf die die Variablen verweisen: Aufgrund möglicher Aliase kann der Zugriff nicht so leicht einem einzigen Objekt vorbehalten werden.



(s. Abschnitte 1.1 und 1.5). Da Instanzvariablen in der Regel Verweise enthalten und Verweise immer den gleichen Platz belegen, ist die Größe eines Objekts (der zu seiner Speicherung benötigte Platz) mit der Anzahl seiner Instanzvariablen festgelegt.

In SMALLTALK werden zwei Arten von Instanzvariablen unterschieden: **benannte** und **indizierte**. Jede benannte Instanzvariable benennt (oder verweist auf) jeweils ein Objekt; der Name der Variable wird somit für die Dauer, die die Variable auf das Objekt verweist, auch zum Namen des Objekts. Da es sich bei Instanzvariablen um lokale Variablen handelt, muss der Name einer benannten Instanzvariablen in SMALLTALK mit einem Kleinbuchstaben beginnen.

Unterscheidung von benannten und indizierten Instanzvariablen

Indizierte Instanzvariablen haben keine Namen, sondern werden über einen Index relativ zu dem Objekt, zu dem sie gehören, angesprochen. Damit ist der Index gewissermaßen der Name der Instanzvariable. Der Index muss eine natürliche Zahl größer 0 sein. Um den Inhalt der indizierten Instanzvariable an der Indexposition **i** (genauer: an der Indexposition, die durch das Zahlobjekt bestimmt wird, auf das **i** verweist) zu erhalten, schreibt man

indizierte Instanzvariablen

33 `at: i`

Wer bei indizierten Instanzvariablen an Arrays denkt, liegt richtig: Tatsächlich speichern Array-Objekte ihre Elemente in indizierten Instanzvariablen. So liefert beispielsweise

34 `#($a $b $c) at: 2`

das Zeichenobjekt „b“. Um den Wert einer indizierten Instanzvariable an derselben Indexposition zu setzen, schreibt man z. B.

35 `#($a $b $c) at: 2 put: 'toll!'`

(aber nur, wenn Ihr SMALLTALK die Änderung der Zusammensetzung für literale Arrays zulässt). Das resultierende Array-Objekt hat die literale Repräsentation `#($a 'toll!' $c)`.

10

Indizierte Instanzvariablen sind kein Unikat von SMALLTALK: So bieten beispielsweise C# und VISUAL BASIC sog. *Indexer*, die im wesentlichen indizierten Instanzvariablen entsprechen (s. Abschnitt 50.3.2 in Kurseinheit 5). Auch verfügen manche Objekte in VISUAL BASIC FOR APPLICATIONS (VBA) über eine Variable **Item**, deren Elemente über Indizierung des Objekts, dem sie zugeordnet ist, angesprochen werden können.

indizierte Instanzvariablen in anderen Sprachen

¹⁰ C# und VISUAL BASIC bieten sog. *Indexer*, die im wesentlichen indizierten Instanzvariablen entsprechen (s. Abschnitt 50.3.2 in Kurseinheit 5). Manche Objekte in VISUAL BASIC FOR APPLICATIONS (VBA) verfügen über eine Variable **Item**, deren Elemente über `<objekt>.Item(n)`, aber auch direkt über `<objekt>(n)` angesprochen werden können.



Die Anzahl der indizierten Instanzvariablen eines Objekts ist fix. Damit ist auch die Größe eines Objekts mit indizierten Instanzvariablen fest; insbesondere können Array-Objekte nicht wachsen (und wenn doch, dann nur über den in Abschnitt 1.1 erwähnten Trick mit dem Wechsel der Identität). Es müssen aber nicht alle indizierten Instanzvariablen belegt sein; die „leeren“ enthalten dann `nil` (s. u.).

**Zuordnung von
Instanzvariablen zu
Objekten**

Es bleibt noch die Frage, wie Objekte in SMALLTALK in den Besitz von Instanzvariablen gelangen. Um das zu erklären, müsste an dieser Stelle auf das Konzept der Klasse vorgegriffen werden, was aber aus didaktischen Gründen unterbleiben soll. Gelernte PASCAL-Programmiererinnen können sich die Instanzvariablen aber wie die Felder eines Records vorstellen (oder C-Programmiererinnen wie die eines Structs), die in der Record-Definition festgelegt werden und die für jede Variable vom Typ dieses Records zur Verfügung stehen. Für alle anderen mag es reichen, sich vorzustellen, jedes Objekt verfüge automatisch über zwei spezielle Variablen, die die Namen und die zugewiesenen Objekte aller seiner Instanzvariablen verwalten. Wie so etwas gehen kann, wird in der nächsten Kurseinheit klarwerden.

2.2 Unterscheidung von :1- und :n-Beziehungen

In der Daten- und Softwaremodellierung werden Beziehungen (oder Relationen) häufig mit sog. *Kardinalitäten* versehen. (Manchmal, besonders im Kontext der Softwaremodellierung mit der Unified Modeling Language *UML* werden diese auch *Multiplizitäten* genannt.) Sie geben an, mit wie vielen anderen Objekten ein Objekt gleichzeitig in derselben Beziehung stehen kann. Beispielsweise kann eine Person zu mehreren anderen Personen in einer Verwandtschaftsbeziehung stehen. Häufig sind die möglichen Kardinalitäten auf ein Intervall beschränkt; sie werden dann durch eben dieses Intervall beschrieben.



Von den theoretisch unendlich vielen möglichen Intervallen, die die Kardinalität beschränken können, sind vor allem drei interessant: [0..1], [1..1] und [0..∞). Dabei ist [1..1], also dass ein Objekt immer mit genau einem in Beziehung stehen muss, technisch nur schwer umzusetzen, so dass [1..1] hier nicht weiter betrachtet wird¹¹; die untere Schranke 0, die den beiden verbleibenden Intervallen gemeinsam ist und die ausdrückt, dass ein Objekt auch mit gar keinem anderen in der Beziehung stehen kann, muss daher nicht erwähnt werden. Im Fall von [0..1] sprechen wir also von **Zu-eins-Beziehungen** (im folgenden mit **:1-Beziehung** notiert), im Fall von [0..∞) von **Zu-n-Beziehungen (:n-Beziehungen)**, wobei *n* hier andeuten soll, dass es sich um eine nicht näher spezifizierte Zahl größer als 1 handelt.¹²

¹¹ Eine Beziehung zu keinem Objekt, wird, im Fall von [0..1], in der Regel durch eine Beziehung zum Objekt `nil` (`null` in anderen Sprachen) ausgedrückt. Nach und nach kommen in verschiedenen objektorientierten Programmiersprachen die sog. Not-null-Annotationen auf, die sicherstellen sollen, dass eine Variable nie den Wert `null` hat.

¹² Aus der relationalen Datenmodellierung sind 1:1-, 1:n- und m:n-Beziehungen bekannt, die Kardinalitäten auch für die Gegenrichtung angeben. Da wir es in der objektorientierten Programmierung



Die Beziehung eines Objekts zu einem anderen wird auf natürliche Weise durch eine benannte Instanzvariable ausgedrückt, wobei die Instanzvariable den Namen der Beziehung oder, besser noch, den Namen der Rolle des von der Variablen referenzierten Objektes in der Beziehung trägt. So zeigt die Instanzvariable **arbeitgeber** beispielsweise auf das Objekt, das in der Beziehung *Anstellung* aus Sicht des Arbeitnehmerobjekts die Rolle des Arbeitgebers spielt. Hat auch das Arbeitgeberobjekt einen Verweis auf das Arbeitnehmerobjekt (die Rückrichtung), so wird die entsprechende Variable sinnvollerweise nach der Gegenrolle **arbeitnehmer** genannt. Steht ein Arbeitnehmerobjekt zur Zeit in keinem Anstellungsverhältnis, ist seine Instanzvariable **arbeitgeber** leer, was in SMALLTALK durch den Verweis auf das Objekt **nil** ausgedrückt wird.¹³

Beziehungen sind nicht von Natur aus auf *ein* Gegenüber eingeschränkt:

Ein Objekt kann, und wird häufig, in derselben Beziehung zu mehreren andern stehen. Genau dafür sind aber die indizierten Instanzvariablen wie geschaffen: Sie erlauben es, von einem Objekt zu beliebig vielen anderen Objekten zu navigieren, ohne für jedes andere eine eigene (jeweils anders) benannte Instanzvariable vorsehen zu müssen. Die „Namen“ der Gegenüber sind einfach Indizes: 1, 2, 3 usw.

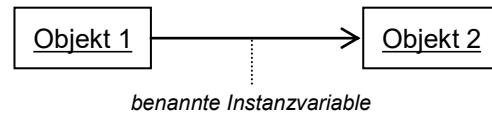
Es ergibt sich nun aber das Problem, dass bei durch indizierte Instanzvariablen eines Objekts realisierten :n-Beziehungen nicht zwischen verschiedenen solchen Beziehungen desselben Objekts unterschieden werden kann — die indizierten Instanzvariablen sind ja nicht benannt. Deswegen werden :n-Beziehungen in der objektorientierten Programmierpraxis praktisch immer über **Zwischenobjekte** realisiert, deren Aufgabe es ist, mittels ihrer indizierten Instanzvariablen jeweils *eine* Beziehung zu mehreren anderen Objekten herzustellen. Dabei können diese Zwischenobjekte die :n-Beziehung ggf. mit weiteren Attributen (z. B. Anzahl *n*, Verweise auf ein bestimmtes Element, Art der Sortierung o. ä.) versehen, die dann in den benannten Instanzvariablen der Zwischenobjekte untergebracht werden. Das Originalobjekt, das die :n-Beziehung eigentlich haben sollte, steht dann stattdessen in einer von einer benannten Instanzvariable hergestellten :1-Beziehung zu dem Zwischenobjekt, das die :n-Beziehung herstellt.

aber nicht mit (ungerichteten, d. h. bidirekationalen) Relationen, sondern mit Verweisen (Zeigern) zu tun haben, entfällt die Rückrichtung.

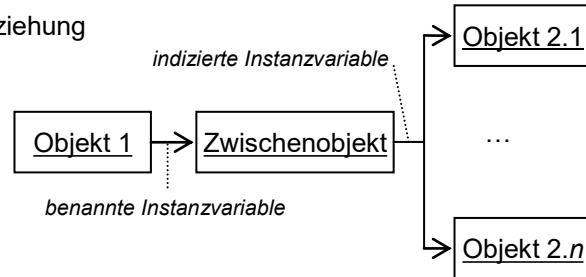
¹³ Diese Konvention wurde vom Turing-Preisträger SIR CHARLES ANTONY RICHARD HOARE eingeführt, der sie heute selbst als einen (seinen) „billion dollar mistake“ bezeichnet.



:1-Beziehung



:n-Beziehung



Wie wir noch sehen werden, erlaubt der Umstand, dass $:n$ -Beziehungen über Zwischenobjekte realisiert werden, die vollwertige Objekte sind, die Beziehungen beliebig auszustalten. So kann beispielsweise eine (Sortier-)Reihenfolge vorgegeben oder ein ausgezeichnetes Element der Beziehung noch einmal gesondert referenziert werden (z. B. das oberste Element auf einem Stack). Auch besondere Zugriffsverfahren wie z. B. das Auffinden von Elementen (in Beziehung stehenden Objekten) anhand eines Schlüssels können auf diese Weise realisiert werden. Da in SMALLTALK Objekte auch eigene Kontrollstrukturen (wie z. B. spezielle Schleifen) anbieten können, sind der Ausgestaltung von Beziehungen über Zwischenobjekte praktisch keine Grenzen gesetzt.

Da $:n$ -Beziehungen häufig vorkommen, ist ihre Handhabung von entscheidender Bedeutung für die Ausdrucksstärke der verwendeten Programmiersprache und die Produktivität der Programmierung insgesamt. Wie sich schon in Abschnitt 4.6.4 zeigen wird, erlaubt die Ausgestaltung von Zwischenobjekten in SMALLTALK Möglichkeiten, die bis heute Vorbildcharakter für andere objektorientierte Programmiersprachen haben.

Vorteile von Zwischenobjekten

effiziente Bearbeitung von $:n$ -Beziehungen entscheidend für die Produktivität

2.3 Teil-Ganzes-Beziehungen

Eine Sonderrolle unter den Beziehungen nimmt die sog. *Teil-Ganzes-Beziehung*, je nach Kontext und Jargon auch *Komposition* oder *Aggregation* genannt, ein. Teil-Ganzes-Beziehungen bestimmen ganz wesentlich unsere Weltsicht: Alles, was wir anfassen oder betrachten können, ist aus kleineren Teilen zusammengesetzt, die selbst wieder Zusammensetzungen (Aggregate, Komposita) sind bis hinunter zu den Elementar-, d. h., unteilbaren Bausteinen.



keine
allgemeingültige
Definition der Teil-
Ganzes-Beziehung



WIKIPEDIA

Nun ist der Begriff der Teil-Ganzes-Beziehung leider nicht so klar definiert, wie es auf den ersten Blick scheint. Tatsächlich bestehen, je nach Art der Zusammensetzung, zum Teil völlig unterschiedliche Wechselwirkungen zwischen dem Ganzen und seinen Teilen. Zudem gibt es neben der physischen Teil-Ganzes-Beziehung auch eine logische: So ist z. B. der Deutsche Bundestag aus einer Anzahl von Abgeordneten zusammengesetzt und jede Familie aus ihren Mitgliedern. Tatsächlich gibt es so viele Varianten der Teil-Ganzes-Beziehung, dass der (philosophische) Diskurs darüber ganze Regale füllt und zu einer eigenen Disziplin geführt hat (der sog. *Mereologie*). Verständlicherweise kann dem eine Programmiersprache nicht folgen und für jede dieser Beziehungen ein eigenes Sprachkonstrukt anbieten.

mangelnde
Sprachunterstützung

Stattdessen bieten die meisten (objektorientierten) Programmiersprachen aber leider überhaupt kein Sprachkonstrukt an, das speziell für die Teil-Ganzes-Beziehung gedacht wäre. Gleichwohl kann man die Unterscheidung zwischen Instanzvariablen mit Referenz- und Wertsemantik, falls vorhanden, dazu nutzen, um zumindest eine spezielle Form der Teil-Ganzes-Beziehung abzubilden: Da bei Wertsemantik mit der Entfernung eines Objekts aus dem Speicher auch alle Objekte, die als Werte seiner Instanzvariablen dienen, aus dem Speicher entfernt werden, kann man hier tatsächlich von der Umsetzung einer bestimmten Form von Teil-Ganzes-Beziehung sprechen, nämlich einer solchen, bei der die Existenz der Teile von der Existenz des Ganzen abhängt (in der UML auch *Komposition* genannt). Für andere Formen, wie z. B. die Bildung einer Gruppe von Objekten als Objekt mit eigener Identität (einem Verein beispielsweise), ist dieses Modell aber nicht geeignet, da sonst mit Auflösung der Gruppe auch die Gruppenmitglieder verschwinden müssten. Für die SMALLTALK-Programmiererin sind diese Überlegungen aber sowieso kein Thema, denn sie hat die Wahl erst gar nicht.

Mit der Teil-Ganzes-Beziehung auf Programmebene werden wir uns in Kurseinheit 6 (genauer: Kapitel 58 und 59) noch ausführlicher beschäftigen. Hier sei nur schon soviel gesagt, dass die Möglichkeit des (rekursiven) Aufbaus eines Software-Systems aus Teilen, die jeweils ihren inneren Aufbau (ihre Komposition) *kapseln*, also insbesondere die Nichtexistenz von Aliassen auf ihre Teile garantieren, genau das ist, was der objektorientierten Programmierung im wesentlichen bis heute fehlt.

2.4 Attribute

Logisch kann man Instanzvariablen in zwei Kategorien aufteilen: solche, die Eigenschaften eines Objekts festhalten, und solche, die tatsächliche Beziehungen zwischen Objekten repräsentieren. Typische Eigenschaften sind beispielsweise die Farbe von etwas oder der Name; sie grenzen sich von Beziehungen inhaltlich dadurch ab, dass das bezogene Objekt isoliert betrachtet seine Bedeutung verliert. So ist das Objekt **rot** allein nichts weiter als eine Farbe — erst als Attribut eines Objekts (wie z. B. „Apfel“) bekommt es eine Bedeutung. Das gleiche gilt für „Schmidtchen“ oder „1“. Dazu kommt, dass Attributwerte wie die vorgenannten in der Regel selbst keine Attribute haben oder Beziehungen mit anderen Objekten



eingehen. Man beachte jedoch, dass dieses Unterscheidungskriterium nicht absolut, sondern relativ zur jeweils betrachteten Domäne ist: Wenn es z. B. um Farben geht, ist „rot“ ein Objekt, das für sich genommen schon eine Bedeutung hat und das mit anderen Objekten vollwertige Beziehungen eingeht, so z. B. mit „grün“ als seinem Komplementärkontrast.

Wenn wir eben von *Attributwerten* sprachen, so ist das nicht ganz zufällig: Nicht selten haben Variablen, die Attribute repräsentieren, zumindest logisch eine Wertsemantik, d. h., sie halten (oder verweisen auf, je nach Implementierung der Sprache) eigene Kopien eines Objekts. Ein typisches Beispiel hierfür hatten Sie in Abschnitt 1.8 bereits kennengelernt: So ist es sinnvoll, dass die Änderung des Namens (genauer: des Namensobjekts) bei einer Person nicht gleichzeitig andere Personen, die den gleichen (nicht denselben!) Namen haben, betrifft. Eine ähnliche Überlegung spielt im Zusammenhang mit der Betrachtung des Zustandes eines Objekts eine Rolle.

Attribute mit Wertsemantik

3 Zustand

Wie bereits in Abschnitt 1.3 beschrieben, kann man zwischen *veränderlichen* und *unveränderlichen Objekten* unterscheiden. Veränderungen veränderlicher Objekte erfolgen über die Zeit und die Objekte wechseln dabei ihren **Zustand**; unveränderliche Objekte dagegen haben keinen Zustand.¹⁴ Was aber macht den Zustand eines Objektes aus?

Der Zustand eines Objektes ist die Summe der Belegungen seiner Instanzvariablen. Insofern Instanzvariablen Beziehungen ausdrücken, wird der Zustand eines Objekts ausschließlich durch seine Verknüpfung mit anderen Objekten definiert. Zudem folgt, dass die einzige Möglichkeit, den Zustand eines Objekts zu ändern, der über die Zuweisung von Instanzvariablen, gleichbedeutend mit der Änderung seiner Beziehungen, ist.

Änderung des Zustands eines Objektes

3.1 Eingrenzung

Eine etwas eingeschränktere Sicht vom Zustand eines Objektes bezieht nur seine *Attributwerte* ein. Dies setzt jedoch voraus, dass überhaupt formal zwischen Attributen und Beziehungen unterschieden werden kann. In Ermangelung spezieller Schlüsselwörter könnte dies, wie bereits oben diskutiert, allenfalls über die Unterscheidung von Variablen mit Wert- und solchen mit Referenzsemantik erfolgen. Dies ist

Attribute und Zustand

¹⁴ Wenn sie einen hätten, dann immer denselben. Da der Begriff des Zustandes jedoch ohne die Möglichkeit der Veränderung kaum einen Sinn hat, sprechen wir auch nicht vom Zustand unveränderlicher Objekte wie z. B. „1“. Manche SMALLTALK-Dialekte sehen jedoch eine Instanzvariable **immutable** vor, die die Veränderlichkeit von Objekten festlegt und die, so ihr Wert denn geändert werden kann, die (Un-)Veränderlichkeit selbst zu einem Teil des Zustandes von Objekten macht.



jedoch schon in Sprachen, die der Programmiererin eine entsprechende Entscheidung anbieten, nicht automatisch der Fall (z. B. JAVA, da hier Strings zwar unveränderlich sind und somit eigentlich zu den Werten zählen, aber dennoch Referenzsemantik haben); in SMALLTALK schließlich ist, da alle Instanzvariablen Referenzen enthalten können, diese Einschränkung überhaupt nicht anwendbar.

Man könnte nun die obige Aussage, dass Zustandsänderungen eines Objekts ausschließlich über Zuweisungen an seine Instanzvariablen erfolgen können, anzweifeln und fragen, ob sich der Zustand eines Objektes auch dann ändert, wenn sich der Zustand eines Objektes, auf das es (per Instanzvariable) verweist, ändert? Zunächst würde man diese Frage mit nein beantworten, denn sonst würde ja, da Objekte in der Regel stark miteinander verflochten sind, die Änderung des Zustandes eines Objektes fast immer zu einer wahren Kettenreaktion führen: Der Zustand aller Objekte, die direkt oder indirekt — also über dritte — darauf verweisen, würde sich mit ändern. Diese Definition von Zustand würde jedoch kaum unserem Weltbild entsprechen: Die Änderung Ihres Familienstandes beispielsweise würde kaum etwas an meinem Zustand ändern, obwohl Sie meine Kurstexte geschickt bekommen, ich also in einer (direkten oder indirekten) Beziehung zu Ihnen stehe. Wie aber ist es, wenn ich meinen Namen ändere? Ändert sich dann mein Zustand?

Reichweite von Zustandsänderungen

Während man diese Frage sicher mit ja beantworten wird, ist doch mein Name als String ein eigenständiges Objekt, das seinen Zustand wechselt, wenn ich, wie im Beispiel von Abschnitt 1.8 geschehen, einzelne Buchstaben in ihm austausche oder ergänze. Nach obiger Auffassung ändert sich mein Zustand (als Besitzer des Namens) dadurch nicht, denn es bleibt ja dasselbe (per Identität) String-Objekt, das meinen Namen hält — es hat lediglich seinen Zustand gewechselt. Etwas anderes wäre es hingegen, wenn ich das String-Objekt, das meinen Namen repräsentiert, gegen ein anderes austausche: Dann verweist die entsprechende Instanzvariable auf ein anderes Objekt und mein Zustand hat sich sicher geändert. Gleichwohl würde man dasselbe auch von der Namensänderung per String-Manipulation erwarten, nur technisch lässt sich dieser Sonderfall nicht begründen! Wie Sie sehen, ist das objektorientierte Denkmodell nicht ganz ohne Tücken, und Programmierfehler schleichen sich an Stellen ein, die so simpel aussehen, dass man dort nie einen Fehler vermuten würde. Vielleicht auch deswegen ist es sinnvoll, wenn Strings, wie von einigen SMALLTALK-Dialektken (und übrigens auch von JAVA) vorgesehen, immer unveränderlich (immutable) sind.

Sonderfall Strings

Auch wenn es nicht sinnvoll ist, für ein allgemeines Objektgeflecht den Zustandsbegriff auf mehrere Objekte auszudehnen, so ist dies für Kompositionen, also aus Teilen zusammengesetzten Ganzen (vgl. Abschnitt 2.3), durchaus angemessen: Wenn z. B. ein Dokument aus mehreren Seiten besteht, die wiederum jeweils aus Zeilen und Spalten bestehen, dann ändert die Änderung einer Zeile auch den Zustand des gesamten Dokuments. Dies wird weiter unten noch eine Rolle spielen, vor allem im Zusammenhang mit Aliasing: Wenn man nämlich davon ausgeht, dass Zustandsänderungen eines Objekts stets Sache des Objekts selbst sind, dann dürfen keine externen Aliase auf seine Teile existieren, denn sonst könnte ihm eine Zustandsänderung von außen quasi untergejubelt werden. Man sollte

Zustand und Komponenten



3.2 Kapselung

Die Unterscheidung von lokalen und globalen Variablen aus Abschnitt 1.5.2 dient u. a. dem Verbergen von Geheimnissen (das sog. **Geheimnisprinzip** engl. auch **Information hiding** genannt), genauer von **Implementationsgeheimnissen**. So ist es fast immer sinnvoll, die Struktur zusammengesetzter Objekte vor den sie benutzenden Objekten zu verbergen, damit man diese Struktur später ändern kann, ohne dass die benutzenden (abhängigen) Objekte davon betroffen wären. Derartige Änderungsabhängigkeiten werden verhindert, wenn die Variablen von außen gar nicht zugreifbar sind, was für lokale Variablen, die ja von außen unsichtbar sind, automatisch der Fall ist.

lokale Variablen und das Geheimnisprinzip

Vom Geheimnisprinzip abzugrenzen ist der Begriff der **Kapselung**, den man mit der objektorientierten Programmierung verbindet: Ein Objekt soll seinen Zustand kapseln, so dass dieser nur von ihm selbst geändert werden kann. Anders als beim Information hiding geht es bei der Kapselung also nicht um Änderungen des Aufbaus von Objekten, sondern um Änderungen ihres Zustandes. Leider lässt sich die Kapselung nicht mit denselben Mitteln wie das Geheimnisprinzip umsetzen: Aufgrund des Aliasing kann ein Objekt, dessen einer Name (beispielsweise aufgrund des Geheimnisprinzips) unsichtbar ist, über einen anderen zugreifbar sein, ohne dass der erste Name etwas dagegen machen könnte. Über lokale Instanzvariablen kann ein Objekt also verbergen, welche Objekte es kennt; es kann aber nicht verhindern, dass andere Objekte diese Objekte auch kennen und, ohne sein Wissen, manipulieren. Es ist somit wegen der etwaigen Existenz von Aliasen nicht möglich, dass ein Objekt seinen inneren Aufbau vor der Außenwelt *abkapselt*, es sei denn, es hat ganz spezielle Vorkehrungen dafür getroffen. Da diese Vorkehrungen (derzeit noch) in keine gängige objektorientierte Programmiersprache eingebaut sind¹⁵, sondern explizit programmiert werden müssen, werden wir uns hier (in dieser Kurseinheit) nicht weiter damit beschäftigen; eine weitergehende Betrachtung erfolgt in Kapitel 58 von Kurseinheit 6).

lokale Variablen und Kapselung

4 Verhalten

Wenn Objekte ihren Zustand kapseln, ist es ausschließlich ihr Verhalten, welches ihn ändert, welches also die Zustandsänderungen eines Objekts herbeiführt. Umgekehrt hängt das Verhalten eines Objekts in der Regel von seinem Zustand ab. Wie aber wird das Verhalten eines Objekts beschrieben? Bevor wir uns dieser Frage zuwenden, müssen wir erst wir zunächst den Begriff des Ausdrucks klären.

¹⁵ Die Programmiersprache Rust ist hier vielleicht eine Ausnahme.



4.1 Ausdrücke

In einem objektorientierten Programm können Objekte nicht nur durch Literale (Abschnitt 1.2) und Variablen (Abschnitt 1.5) repräsentiert werden, sondern auch durch **Ausdrücke**. Tatsächlich sind Literale und Variablen *primitive Ausdrücke*, also solche, die nicht aus anderen zusammengesetzt sind. Damit mit den Objekten aber etwas geschehen und ein Programm somit etwas tun kann, brauchen wir noch andere Ausdrücke, namentlich *Zuweisungsausdrücke* und *Nachrichtenausdrücke*. Auch sie stehen jeweils für ein Objekt und können deswegen überall da auftreten, wo Objekte verlangt werden. Sie können insbesondere auch geschachtelt werden.

4.1.1 Zuweisungsausdrücke

Zuweisungsausdrücke hatten Sie schon in Abschnitt 1.6 kennengelernt. Sie verlangen auf der linken Seite eine Variable und auf der rechten einen Ausdruck:

36 `x := 1`

etwa ist ein Zuweisungsausdruck. Wie bereits erwähnt, bewirken Zuweisungen als einzige Ausdrücke den Zustandswechsel von Objekten. So bewirkt etwa

Zustandswechsel von Objekten durch Zuweisungen

37 `name := 'Hänschen'`

wobei `name` eine Instanzvariable sein soll, die Änderung des Zustandes des Objektes, zu dem die Instanzvariable gehört.

Da eine Zuweisung selbst ein Ausdruck ist, kann sie auf der rechten Seite einer anderen Zuweisung erscheinen:

38 `y := x := 1`

ist also ein legaler Ausdruck (zu seiner Auswertung kommen wir später, wenn wir — in Abschnitt 4.1.4 — über die Reihenfolge der Auswertung geschachtelter Ausdrücke sprechen).

4.1.2 Nachrichtenausdrücke

Neben der Zuweisung ist der **Nachrichtenversand** die zweite wichtige **Ausdrucksform** der objektorientierten Programmierung. SMALLTALK verwendet hierfür eine Syntax, die stark an die der englischen Sprache angelehnt ist: Sie verlangt ein Subjekt (den Empfänger der Nachricht), ein Prädikat (die **Nachricht**) sowie eine optionale Liste von Objekten als Prädikatsergänzungen (die Parameter der Nachricht). Dabei wird auf die in anderen Sprachen übliche Verwendung des Punkts als Trennzeichen zwischen Empfänger und Nachricht und Klammern zum Umschließen der Parameterliste verzichtet; stattdessen verwendet man bei



zwei oder mehr Parametern Partikeln (Präpositionen oder Konjunktionen) ähnelnde Nachrichtenteile, die den Parametern vorangestellt werden. Die allgemeine Syntax lautet also

39 <Empfänger> <Nachricht>

für parameterlose Nachrichten,

40 <Empfänger> <Nachricht> <Parameter>

für Nachrichten mit einem Parameter,

41 <Empfänger>
42 <NachrichtTeil1> <Parameter1>
43 <NachrichtTeil2> <Parameter2>

für Nachrichten mit zwei Parametern usw., wobei hier die in spitzen Klammern stehenden Namen *metasyntaktische Variablen*, also Platzhalter für Namen in einem konkreten Programm, sind und die Nachrichten(teile), die Parameter nach sich ziehen, immer mit einem Doppelpunkt enden müssen.

44 einObjekt tueEtwas

drückt beispielsweise den Versand einer parameterlosen Nachricht **tueEtwas** an das Objekt, auf das die Variable **einObjekt** verweist, aus.¹⁶ Ein etwas konkreteres Beispiel hierfür ist der Ausdruck

45 #abc printString

mit dem dem Objekt **#abc** die Nachricht **printString** gesendet wird.

Soll einer Nachricht ein Parameter angehängt werden, so tut das

46 einObjekt tueEtwasMit: einemParameter

wobei **einemParameter** hier auch eine Variable ist (es könnte auch ein anderer Ausdruck dort stehen; s. u.). Ein konkretes Beispiel hierfür ist

47 Transcript show: 'Hallo'

Ein zweiter Parameter wird durch einen weiteren Nachrichtenbestandteil wie in

48 einObjekt tueEtwasMit: einemParameter und: zweitemParameter

hinzugefügt und alle weiteren entsprechend, also etwa

¹⁶ Es ist übrigens Sitte (aber keineswegs verpflichtend), in SMALLTALK die Zusammensetzung von Namen durch sog. Binnenmajuskeln (eingefügte Großbuchstaben) und nicht durch den Unterstrich kenntlich zu machen.



```
49 einObjekt
50   tueEtwasMit: einemParameter
51   und: zweitemParameter
52   und: drittem Parameter
```

wobei sich die Nachrichtenteile wie oben ruhig wiederholen dürfen. Es ist also insbesondere nicht so, dass die Reihenfolge der Nachrichtenteile beliebig umgestellt werden dürfte, ohne dass sich dadurch die Bedeutung der Nachricht änderte. Tatsächlich ist die an das Objekt geschickte Nachricht, der sog. **Nachrichtenselektor** (engl. message selector), immer *ein* Symbol, das aus der Konkatenation (Aneinanderfügung) aller seiner Teile, also im Beispiel der Zeile 48 oben `#tueEtwasMit:und:` besteht. Die Namen der Nachrichtenteile sind frei wählbar, beginnen jedoch per Konvention mit einem Kleinbuchstaben.

Nachrichtenselektor

Parameterlose Nachrichten wie beispielsweise `tueEtwas` in Zeile 44 nennt man in SMALLTALK übrigens **unär** (unary messages): Obwohl sie keine expliziten Argumente haben, heißen sie trotzdem unär, weil der Empfänger das erste, implizite Argument ist. So nehmen denn auch sog. **binäre Nachrichten**¹⁷ nur einen Parameter, was aber zwei Argumenten, nämlich dem Empfänger und einem weiteren Argument, entspricht:

unäre und binäre Nachrichten

53 1 + 2

beispielsweise drückt aus, dass die Nachricht „+“ mit Argument „2“ an das Objekt „1“ gesendet werden soll.

In SMALLTALK sind binäre Nachrichten eine syntaktische Besonderheit: Sie bestehen nämlich aus einem oder mehreren nicht alphanumerischen, nicht reservierten Zeichen (die Liste der reservierten Zeichen finden Sie in Kapitel 5 am Ende dieser Kurseinheit). Alle anderen Nachrichten, die neben dem Empfänger noch mindestens ein Argument erfordern, werden in SMALLTALK dagegen **Schlüsselwortnachrichten** (keyword messages) genannt, so z. B. `tueEtwasMit:` in Zeile 46 und `tueEtwasMit:und:` in Zeile 48. Dies ist jedoch etwas verwirrend, da sie keine Schlüsselwörter im landläufigen Sinne (von denen SMALLTALK ja gar keine hat) enthalten und da natürlich auch die unären Nachrichten „Schlüsselwörter“ verwenden.

Ein Nachrichtenausdruck besteht also aus einem Empfängerobjekt, einem Nachrichtenselektor sowie einer Anzahl von Argumentausdrücken, die die Teile des Nachrichtenselektors sperren (dazwischen stehen).¹⁸ Der Ausdruck als ganzes

Sender und Empfänger

¹⁷ nicht zu verwechseln mit den sog. *binären Methoden*, bei denen Empfänger und Parameter den gleichen Typ haben

¹⁸ Nur in Ausnahmefällen ist es sinnvoll, das Empfängerobjekt mit einer Nachricht als (zusätzlichen) Parameter zu übergeben; in den allermeisten Fällen handelt es sich dabei um einen Anfängerinnen-



steht für ein Objekt, nämlich das Ergebnis der Auswertung der Nachricht durch den Empfänger. Der Nachrichtensender wird dem Empfänger übrigens nicht mitgeteilt (es sei denn, er wird explizit als Parameter übergeben) — das System weiß automatisch, wohin die Antwort auf die Nachricht zurückgeliefert werden soll (nämlich genau an die Stelle, an der der Nachrichtenversand steht). Mehr dazu gleich, wenn es um die Auswertung von Nachrichtenausdrücken geht (Abschnitt 4.1.3).

Da ein Nachrichtenausdruck für ein Objekt steht, kann er selbst Teil eines Nachrichtenausdrucks sein, also für den Empfänger der Nachricht oder einen ihrer Parameter stehen. Es ist dann allerdings u. U. notwendig, den so geschachtelten Nachrichtenausdruck zu klammern, da er sonst nicht richtig erkannt werden kann:

geschachtelte Nachrichtenausdrücke

```
54 aStack
55   push: ((aStack pop)
56     arg: (aStack pop)
57     arg: (aStack pop))
```

beispielsweise schiebt das Objekt auf den Stack, das Resultat der Nachricht `arg:arg:` gesandt an das oberste Objekt des Stacks mit seinem zweiten und dritten als Parameter ist. (Zu den genauen Regeln zur Reihenfolge der Auswertungen siehe Abschnitt 4.1.4.)

Nicht selten möchte man eine Serie von Ausdrücken an dasselbe Empfängerobjekt senden. SMALLTALK sieht dafür mit der Kaskadierung eine nette syntaktische Abkürzung vor, die es erlaubt, bei einer Sequenz von Nachrichten an dasselbe Objekt dieses nicht immer wiederholen zu müssen. So sorgt beispielsweise

kaskadierte Nachrichtenausdrücke

```
58 Transcript show: '1'; show: #+; show: '2'; cr.
```

anstelle des wesentlich wortreichereren

```
59 Transcript show: '1'.
60 Transcript show: #+.
61 Transcript show: '2'.
62 Transcript cr
```

dafür, dass die Objekte „1“, „+“ und „2“ nacheinander als Parameter der Nachricht `show:` an das von der globalen Variable `Transcript` benannte Objekt, eine Art Systemkonsole, gesendet werden. (Auf die Bedeutung des „..“ kommen wir in Kapitel 4.2 zu sprechen.) Das Beispiel ist übrigens nicht besonders zwingend, da man in SMALLTALK genauso gut auch hätte

```
63 Transcript show: '1', #+, '2'; cr
```

fehler, bei dem durchscheint, dass die Programmiererin (die vermutlich noch in Kategorien der prozeduralen Programmierung denkt) nicht verstanden hat, dass jede Nachricht einen Empfänger hat, ein Nachrichtenausdruck diesen also schon beinhaltet.



schreiben können (wobei `,` ein binärer Nachrichtenselektor ist, der für die Konkatenation steht). `cr` ist übrigens eine (unäre) Nachricht, die für einen Wagenrücklauf (carriage return) auf dem Transcript sorgt.

Man kann sich aber durch Verwendung der Kaskadierung häufig die Einführung einer *temporären Variablen* (s. Abschnitt 4.3) ersparen, und zwar immer dann, wenn anstelle einer Variable als Empfänger (**Transcript** in obigem Beispiel) ein Ausdruck steht, der das Empfängerobjekt liefert und der nur einmal ausgewertet werden soll. Anstelle von

Kaskadierung
anstelle temporärer
Variablen

```
64 eineVariable := <irgendein Ausdruck>.  
65 eineVariable <eine Nachricht>.  
66 eineVariable <noch eine Nachricht>
```

kann man also

```
67 <irgendein Ausdruck> <eine Nachricht>; <noch eine Nachricht>
```

schreiben, wenn man `eineVariable` hernach nicht mehr benötigt. SMALLTALK unterstützt also von Haus aus sog. *Fluent APIs*.



WIKIPEDIA

4.1.3 Auswertung von Ausdrücken

Ausdrücke allein sind nur syntaktische Figuren — um das Objekt zu liefern, für das sie stehen, müssen sie ausgewertet werden. Es ist die Auswertung, bei der ein Programm tatsächlich „etwas tut“.

Das Elementarste, das ein Programm tun kann, ist die Auswertung einer Zuweisung wie beispielsweise `y := x`. Diese Auswertung führt dazu, dass die Variable `y` nach der Zuweisung auf dasselbe Objekt verweist wie vorher schon `x`. Dabei steht der gesamte Ausdruck `y := x` für (eine Referenz auf) das Objekt, auf das `x` (und nach seiner Auswertung auch `y`) verweist. Die Zuweisung dieses Objektes an `y` ist gewissermaßen nur ein *Seiteneffekt* der Auswertung des Ausdrucks.¹⁹

Auswertung von
Zuweisungs-
ausdrücken

Man beachte, dass es sich beim *Zuweisungsoperator* `:=` nicht um einen (binären) Nachrichtenselektor handelt: Es wird hier nämlich keine Nachricht an ein Objekt geschickt, sondern der Inhalt einer Variable manipuliert. Die Zuweisung ist tatsächlich eines der wenigen Primitive SMALLTALKs, also eine der Operationen, die nicht in sich selbst programmiert, sondern fest „verdrahteter“ Teil der Sprache sind (vgl. Abschnitt 1.6).

¹⁹ Über das Wort *Seiteneffekt* wird viel gestritten. Manche meinen, es müsste *Nebeneffekt* heißen, aber dieses Wort ist genauso unnötig, wenn *Nebenwirkung* das ist, was gemeint ist. Als strittig würde ich schon eher ansehen, ob man hier überhaupt von einer Nebenwirkung sprechen sollte, denn der sich einstellende Effekt ist ja das vornehmliche Ziel einer Zuweisung, so dass man eher von Wirkung (oder Effekt) sprechen sollte. Für mich ist *Seiteneffekt* Jargon, bei dem jede weiß, was gemeint ist.



Nachrichtenausdrücke werden ausgewertet, in dem die Nachricht (das Prädikat des Satzes) an das Empfängerobjekt (das Subjekt) mit den Parametern (den Prädikatsergänzungen) gesendet wird und das Empfängerobjekt als Ergebnis der Nachricht ein Objekt zurückliefert. Wie das Empfängerobjekt das Ergebnisobjekt bestimmt, darauf kommen wir noch; hier ist nur wichtig, dass der Nachrichtenausdruck im Zuge der Auswertung gewissermaßen durch das Objekt, für das er steht, ersetzt wird. Man kann sich das genauso vorstellen wie die Auswertung einer (mathematischen) Funktion f , deren Funktionsanwendung $f(x)$ auf ein Objekt x ebenfalls für den Funktionswert steht.

Eine häufig gebrauchte, von allen Objekten verstandene unäre Nachricht ist **printString**, in Reaktion auf die das Empfängerobjekt eine textuelle Repräsentation von sich zurückgibt:

Beispiele

68 **12 printString**

liefert den String '12', die Auswertung von Zeile 45 den String 'abc'. Der bereits in Zeile 53 angeführte binäre Nachrichtenausdruck

69 **1 + 2**

liefert wie erhofft das Objekt „3“. Bei der Auswertung des Schlüsselwortausdrucks

70 **12 printOn: Transcript**

bei dem als Seiteneffekt die Zahl „12“ in ihrer textuellen Repräsentation, also als String '12' auf dem Ausgabestrom, auf den die Variable **Transcript** verweist, ausgegeben wird, wird das Empfängerobjekt „12“ zurückgeliefert.

Es ist in SMALLTALK, anders als z. B. bei als **void** deklarierten Methoden in JAVA oder C#, nicht möglich, in Reaktion auf einen Nachrichtenversand nichts zurückzugeben. Dies könnte man in Zeile 70 als sinnvoll erachten, da man hier eigentlich nur an dem (*Seiten-*) Effekt, der Ausgabe von „12“ auf dem Transcript, interessiert ist. Wenn die zu einer Nachricht gehörende Methode keinen sinnvollen Rückgabewert hat, wird *standardmäßig immer das Empfängerobjekt* zurückgegeben; auch das dient, da man Nachrichtenausdrücke so stets verketten kann, einem *Fluent API*. Sollte es für das Rückgabeobjekt im Kontext der Nachricht (des Nachrichtenverands) keine Verwendung geben, es also nicht einer Variable zugewiesen oder sonst wie Teil eines anderen Ausdrucks sein, verfällt es einfach. Da der Rückgabewert aber technisch nur eine Referenz auf das Objekt ist, bleibt die Existenz des Objektes davon zunächst unberührt. Nur wenn es sonst keine anderen Referenzen auf das Objekt gibt, etwa weil es extra für die Rückgabe erzeugt wurde, wird das Objekt bei der nächsten *Speicherbereinigung* entfernt.

Nachrichten-
ausdrücke als
Funktions-
anwendungen



Da die Auswertung eines Nachrichtenausdrucks zur Abarbeitung der Anweisungen des Rumpfs einer Methode führt, nennt man ihn auch *Methodenaufruf*. Warum diese Bezeichnung legitim ist, wird jedoch erst in Abschnitt 4.3 klar.

Zusammenfassend wird also

- ein Literal zu dem Objekt ausgewertet, das es repräsentiert,
- eine Variable zu dem Objekt, das sie benennt,
- eine Zuweisung zu dem Objekt, zu dem der Ausdruck auf der rechten Seite der Zuweisung ausgewertet wird,
- ein Nachrichtenausdruck zu dem Objekt, das als Antwort auf die Nachricht zurückgegeben wird, sowie
- ein kaskadierter Nachrichtenausdruck zu dem Objekt, zu dem sein letzter Nachrichtenausdruck ausgewertet wird.

4.1.4 Reihenfolge der Auswertung von geschachtelten Ausdrücken

Da Ausdrücke andere Ausdrücke enthalten können, stellt sich die Frage nach der Reihenfolge der Auswertung von geschachtelten Ausdrücken. Diese wird, wie in anderen Sprachen auch, implizit über Präzedenzen und explizit über Klammern geregelt.

Bei der doppelten Zuweisung in Zeile 38 oben ist zunächst vielleicht nicht klar, welche der beiden Zuweisungen zuerst ausgewertet (ausgeführt) werden soll: `y := x` oder `x := 1`. Wenn man jedoch weiß, dass es sich bei `y := x := 1` um einen geschachtelten Ausdruck handelt und jeder (Teil-)Ausdruck für ein Objekt steht, dann muss der zweite Ausdruck zuerst ausgewertet und durch das Ergebnis, „1“, ersetzt werden, denn andernfalls wäre die „1“ dem Ergebnis von `y := x` zuzuweisen, was aber nach Abschnitt 4.1.3 keine Variable, sondern ein Objekt ist. Zeile 38 weist also zuerst `x` und dann `y` das Objekt „1“ zu — die Auswertung erfolgt von rechts nach links.

Auswertung von links nach rechts

Das ist allerdings ein Sonderfall. Grundsätzlich werden in SMALLTALK nämlich alle Ausdrücke von links nach rechts ausgewertet. Dabei haben allerdings unäre Ausdrücke Vorrang vor binären und diese wiederum Vorrang vor Schlüsselwortnachrichten, so dass nur bei gleichrangigen Ausdrücken die Auswertung von links nach rechts erfolgt. So wird beispielsweise in

71 `x + 1 < y`

zunächst die Nachricht „+“ an das Objekt, auf das `x` verweist, mit Argument 1 geschickt und an das Ergebnis die Nachricht „<“ mit `y` als Parameter. Umgekehrt wird bei

72 `x < y - 1`



zunächst der Vergleich angestellt und an das Ergebnis, eines der beiden Objekte „true“ und „false“, die Nachricht „–“ mit Argument 1 geschickt. Jedoch wird der Wahrheitswert (das Wahrheitsobjekt) die Nachricht nicht verstehen und entsprechend mit einer Meldung wie „Nachricht nicht verstanden“ reagieren. Um die Präzedenz zu ändern, kann man einfach Klammern setzen:

```
73 x < (y - 1)
```

hat das gewünschte Ergebnis. Individuelle Operatorpräzedenzen (wie z. B. Punktrechnung vor Strichrechnung) kennt SMALLTALK nicht.

Bei mehrstelligen Nachrichten mit Schlüsselwörtern werden alle Schlüsselwörter eines Ausdrucks als zu einer Nachricht gehörig interpretiert, es sei denn, es wurden Klammern gesetzt. Der Ausdruck

mehrstellige
Schlüsselwort-
nachrichten

```
74 einEmpfänger  
75   tueDiesMitDem: erstesArgument  
76   nachdemDuDasGetanHast: zweitesArgument
```

wird also als *eine* Nachricht interpretiert; falls

```
77 einEmpfänger  
78   tueDiesMitDem: (erstesArgument  
79           nachdemDuDasGetanHast: zweitesArgument)
```

gemeint gewesen sein sollte, müssen eben die Klammern entsprechend gesetzt werden. (Man beachte, dass im geklammerten Ausdruck **erstesArgument** Empfänger der Nachricht **nachdemDuDasGetanHast**: mit **zweitesArgument** als Parameter ist. Das Ergebnis der Auswertung dieses Ausdrucks ist dann Parameter der Nachricht **tueDiesMitDem**: an **einEmpfänger**.)

4.2 Anweisungen

Anweisungen spezifizieren die Schritte, in denen ein Programm ausgeführt wird. In SMALLTALK sind alle Ausdrücke, die nicht Bestandteil von anderen Ausdrücken sind, Anweisungen. Während ihnen in JAVA und C# dazu noch ein Semikolon hintangestellt werden muss, ist Vergleichbares in SMALLTALK nicht nötig.

Folgen von Anweisungen werden in SMALLTALK durch einen Punkt getrennt. Dabei handelt es sich (genau wie in PASCAL oder EIFFEL) um ein Trennzeichen und nicht (wie in den von C abgeleiteten Sprachen wie JAVA oder C#) um einen Teil der Anweisung selbst. Der Punkt am Ende einer Anweisung darf also fehlen, wenn keine weitere Anweisung folgt. Ansonsten entspricht die Wahl des Punktes dem Vorsatz SMALLTALKs, der natürlichen Sprache möglichst ähnlich zu sein. So ist auch die Wahl des Semikolons zur Kaskadierung von Nachrichtenausdrücken zu sehen.

der Punkt als
Trennzeichen



Die einzige andere Form der Anweisung in SMALLTALK ist die **Return-Anweisung**. Auf sie werden wir im Zusammenhang mit Methoden im nächsten Kapitel noch ausführlicher zu sprechen kommen. Sie besteht in SMALLTALK aus dem Sonderzeichen `^` (ursprünglich `↑`, jedoch genau wie `←` auf den meisten Tastaturen nicht verfügbar), gefolgt von einem Ausdruck. Die Return-Anweisung „retourniert“ das Objekt, zu dem dieser Ausdruck auswertet.

Da alle anderen Anweisungen Ausdrücke sind, die zu einem Objekt auswerten, brauchen Methoden (Abschnitt 4.3) und Blöcke (Abschnitt 4.4) in SMALLTALK keine Return-Anweisungen, um ein Objekt zurückzuliefern; sie liefern dann das Objekt, zu dem die letzte Anweisung auswertet.

4.3 Methoden

Die Auswertung von Nachrichtenausdrücken, also von Nachrichten, die an Objekte verschickt werden, erfolgt mit Hilfe sog. **Methoden**. Was ein Objekt in Reaktion auf den Erhalt einer entsprechenden Nachricht tun soll, wird durch eine **Methodendefinition** beschrieben. Eine Methodendefinition besteht dazu aus einem *Methodenkopf*, der in SMALLTALK auch als *Message pattern*, allgemein (und im folgenden) aber eher als **Methodensignatur** bezeichnet wird, einer optionalen Liste von *lokalen Variablen* und einem *Methodenrumpf*. Letzterer enthält die **Anweisungen**, die die Methode ausmachen, die also zur Auswertung eines Nachrichtenausdrucks ausgeführt werden. Es ist üblich, jede Methodendefinition mit einem in doppelte Anführungsstriche gesetzten Kommentar zu versehen, der beschreibt, was diese Methode macht.

Die Methodensignatur besteht aus dem Namen der Methode und der Liste ihrer **formalen Parameter**. Formale Parameter sind *lokale Variablen*, denen beim Aufruf der Methode (s. Abschnitt 4.3.2) automatisch ein Wert, der sog. **tatsächliche Parameter**²⁰, zugewiesen wird und deren Sichtbarkeit auf die Methode, in deren Methodensignatur sie vorkommen, beschränkt ist. Wie alle lokalen Variablen müssen sie mit einem Kleinbuchstaben beginnen. Außerdem sind sie *temporäre Variablen*, was soviel heißt wie dass sie nur für die Dauer der Ausführung der Methode existieren. Damit sind auch die Aliase, die durch formale Parameter gebildet werden können, immer temporär. In SMALLTALK sind formale Parameter zudem *Pseudovariablen* (s. Abschnitt 1.7), d. h., es kann ihnen innerhalb der Methode, für die sie sichtbar sind, nichts zugewiesen werden. Syntaktisch unterscheidet sich SMALLTALK von den meisten anderen Programmiersprachen auch bei der Methodendefinition dadurch, dass die formalen Parameter nicht

**formale und
tatsächliche
Parameter**

²⁰ Der im Englischen gebrauchte Term „actual parameter“ wird auch manchmal mit „aktueller Parameter“ ins Deutsche übersetzt, was oft als falsch verhöhnt wird, aber die Sache trotzdem trifft, zumindest, wenn man die zeitliche Dimension, die mit der Programmausführung einhergeht, in Betracht zieht: Es gibt nämlich in der Regel viele tatsächliche Parameter zu einem Methodenaufruf, von denen — zu jedem Zeitpunkt — höchstens einer der aktuelle ist.



durch Komma getrennt in einer in Klammern eingeschlossenen Liste hinter dem Methodenamen stehen, sondern jeder für sich von einem Nachrichtenbestandteil eingeleitet wird.

Eine Methodendefinition folgt also dem Schema

Schema der Methodendefinition

```
80 <Methodensignatur>
81   "<Kommentar>" 
82   | <Liste lokaler Variablen> |
83   <Folge von Anweisungen>
```

(in spitzen Klammern wieder *metasyntaktische Variablen*, also Platzhalter für entsprechende Programmelemente). Eine einfache Methode wäre etwa

```
84 helloWorld
85   "das berühmt-berüchtigte"
86   | |
87   Transcript show: 'Hello World!'; cr
```

oder, wer es generischer mag,

```
88 sag: einenText
89   "nicht so unflexibel"
90   | |
91   Transcript show: einenText; cr
```

Die Methodensignatur als Teil einer Methodendefinition wird im folgenden immer fett hervorgehoben.

Die Methodensignatur dient der Auswahl der zu einer Nachricht passenden Methode; sie ist das Gegenstück zum *Nachrichtenselektor* aus Abschnitt 4.1.2, anhand dessen die Auswahl der zu einem Nachrichtenausdruck passenden Methode durchgeführt wird. Anders als ein Nachrichtenausdruck nennt eine Methodensignatur aber kein Empfängerobjekt und die offenen Stellen eines Nachrichtenselektors werden ausschließlich durch Variablen, eben die *formalen Parameter*, und nicht durch beliebige Ausdrücke besetzt. Typische Methodensignaturen sind z. B.

```
92 printString
```

für Methoden ohne Parameter,

```
93 + einInteger
```

für binäre Methoden²¹ (mit **einInteger** als formalem Parameter) und

²¹ Der Begriff der binären Methode bezeichnet hier zweistellige Methoden, die eine Sonderzeichenfolge als Name verwenden. In der Literatur wird er häufig anders gebraucht: Der Empfänger und der Parameter einer binären Methode haben denselben Typ (vgl. Fußnote 17 und Abschnitt 29.5).



```
94 printOn: einStrom
```

für alle anderen Methoden mit einem oder mehreren Parametern (in diesem Fall mit nur einem formalen Parameter, `einStrom`). Die Methodensignaturen „passen“ übrigens jeweils zu den entsprechenden Nachrichtenausdrücken aus den Zeilen 68, 69 und 70. Wenn der Nachrichtenselektor aus mehreren Teilen besteht, werden diese (entsprechend den Beispielen in den Zeilen 48 ff.) einfach angehängt.

Wie bereits erwähnt, besteht der Methodenrumpf aus einer Folge von Anweisungen, Ausdrücken, die jeweils durch einen Punkt getrennt sind. Wenn die Anweisungen nichts anderes vorsehen, wird die Ausführung einer Methode nach Abarbeitung der letzten Anweisung explizit mit der Rückgabe des Empfängerobjekts an den Sender der Nachricht beendet. Für explizite Beendigungen und Rückgabe eines anderen Objekts als des Empfängers ist die *Return-Anweisung* (Abschnitt 4.2) vorgesehen.

Rückkehr von einer
Methode und
Rückgabe eines
Wertes

```
95 helloWorld  
96 Transcript show: 'Hello World!'; cr  
97 ^ 'Hello World!'
```

Eine Return-Anweisung darf an beliebigen Stellen innerhalb der Methode auftreten. Die Abarbeitung der Methode kann demnach auch vor Erreichen der textuell letzten Anweisung beendet werden. Die Return-Anweisung beeinflusst also den *Kontrollfluss* des Programms. Wichtig ist, dass eine Methode immer ein Objekt zurückgibt; ein Nachrichtenausdruck (oder *Methodenaufruf*; s Abschnitt 4.3.2) steht als Ausdruck also immer für ein Objekt. Prozeduren im Sinne PASCALS oder Void-Methoden im Sinne von C, JAVA usw. gibt es in SMALLTALK nicht (vgl. a. Abschnitt 4.1.3).

explizite Beendigung
einer Methode

Sollte eine Methode zur Durchführung ihrer Berechnungen *temporäre Variablen* benötigen, so müssen diese zu Beginn der Methode (nach der Methodensignatur und vor der ersten Anweisung) deklariert werden. Die Werte dieser Variablen, die standardmäßig mit `nil` initialisiert werden, sind außerhalb der Methode nicht sichtbar; die Variablen werden insbesondere nach Abarbeitung der Methode vom System wieder entfernt. Sie können sich also auch zwischen zwei Ausführungen einer Methode nichts merken.

temporäre Variablen

Temporäre Variablen können auch der besseren Lesbarkeit dienen, indem sie Zwischenergebnissen einen Namen geben:

```
98 helloWorldX2  
99 | einmal zweimal |  
100 einmal := 'Hello World!'.  
101 zweimal := einmal , einmal.  
102 Transcript show: zweimal; cr
```

Umgekehrt können temporäre Variablen, die nur einmal verwendet werden, eingespart werden, indem man kaskadierte Nachrichtenausdrücke (Abschnitt 4.1.2) verwendet.



Methoden sind die Einheiten des Programms, in denen Sie als Programmiererin Ihre Anweisungen unterbringen. Sie werden nach der Eingabe (und bei jeder Änderung) mit dem „Speichern“ kompiliert („Speichern“ deswegen in Anführungsstrichen, weil Methoden nicht in Dateien gespeichert werden, sondern in einer Datenstruktur SMALLTALKs, und zwar in Form von Objekten). Tatsächlich besteht der Löwenanteil eines jeden SMALLTALK-Programms, ja des gesamten SMALLTALK-Systems, aus Methodendefinitionen. Die Methodendefinitionen entsprechen im wesentlichen der Definition von Funktionen (oder, mit obiger Einschränkung, Prozeduren) in anderen Sprachen; jedoch ist es in SMALLTALK nicht möglich (und in der objektorientierten Programmierung allgemein nicht üblich), Methoden zu schachteln, also eine Methode innerhalb einer anderen zu deklarieren. Außerdem gibt es in SMALLTALK keine „Hauptmethode“ wie etwa die Main-Methoden in der C-Sprachfamilie. Sie müssen dem SMALLTALK-System schon sagen, welche Methode Sie ausgeführt haben wollen, z. B. indem Sie einen entsprechenden Ausdruck eingeben und auswerten lassen.

4.3.1 Zuordnung von Methoden zu Objekten

Methoden sind immer Objekten, den sogenannten Empfängerobjekten, zugeordnet. Wir wollen uns in dieser Kurseinheit noch nicht darauf festlegen, wie diese Zuordnung erfolgt; Sie dürfen jedoch davon ausgehen, dass jedes Objekt über einen Katalog von Methoden verfügt und damit auf die entsprechenden Nachrichten reagieren kann. Diesen Katalog nennt man auch das **Protokoll** eines Objekts (s. Abschnitt 4.3.8).

Die Zuordnung von Methoden zu Objekten erlaubt, dass die Methoden auf die Instanzvariablen des jeweiligen Objekts zugreifen können. Da der Zustand eines Objekts durch die Belegung seiner Instanzvariablen repräsentiert wird (s. Kapitel 3) und weil in den Methoden das Verhalten eines Objektes spezifiziert ist, ergibt sich, dass das Verhalten vom Zustand des Objekts abhängen (durch Berücksichtigung seiner Instanzvariablen) und es ihn gleichzeitig beeinflussen (durch Zuweisungen an die Instanzvariablen) kann. Da Instanzvariablen lokale Variablen sind (lokal zum besitzenden Objekt), sind Methoden sogar die einzige Stelle, an der der Zustand von Objekten geändert werden kann. Mehr dazu in Abschnitt 4.3.4.

Zugriff auf Instanzvariablen

Neben den *formalen Parametern* der Methode und den Instanzvariablen des Empfängerobjekts ist im Rumpf jeder Methode eine weitere Variable zugreifbar, die den Namen „**self**“ trägt. Diese Variable, die wie die formalen Parameter eine *Pseudovariable* ist, verweist immer auf das Empfängerobjekt der Nachricht, also auf das Objekt, dessen Instanzvariablen gerade zugreifbar sind. Sie wird immer dann benötigt, wenn eine Nachricht aus einer Methode heraus (also per darin enthaltener Anweisung) an das Objekt geschickt werden soll, dem die Methode zugeordnet ist, also an sich selbst. **self** ist also gewissermaßen der implizite erste Parameter einer Methode.

die Pseudovariable **self**



4.3.2 Methodenaufruf und dynamisches Binden

Wenn das Versenden von Nachrichten bislang als die Übergabe eines entsprechenden Nachrichtenobjekts an den Empfänger dargestellt wurde, so ist das nicht ganz richtig: Vielmehr wird ein Nachrichtenausdruck auch in SMALLTALK aus Effizienzgründen vom Compiler in einen schnöden **Methodenaufruf** übersetzt, der mit dem Funktionsaufruf aus der prozeduralen Programmierung (also z. B. PASCAL oder C) vergleichbar ist. So führt beispielsweise der Ausdruck

103 1 + 2

zum Aufruf der Methode `+`, wie sie für das Objekt „1“ (und alle Zahlen) definiert wurde. Dennoch wird, wohl aus didaktischen Gründen, das Mysterium vom Nachrichtenversand in der objektorientierten Literatur weiter gepflegt. Es gibt aber auch einen kleinen, feinen Unterschied zum gewöhnlichen Prozederaufruf.

Die Entscheidung, welche Methode in Reaktion auf einen Nachrichtenversand aufgerufen und abgearbeitet wird, hängt nicht von dem Nachrichtenselektor allein ab, sondern auch von dem Objekt, an das die Nachricht geschickt wird. Es ist nämlich durchaus üblich, dass verschiedene Objekte mit gleichen Methodensignaturen unterschiedliche Methodenimplementierungen verbinden; so implementieren beispielsweise Zahlen und Symbole die Methode `printString` jeweils anders und selbst

Abhängigkeit vom Empfängerobjekt

104 1.0 + 2

führt zum Aufruf einer anderen Methode als `1 + 2`,

105 1 + 2.0

dagegen nicht.

Aus der Abhängigkeit des Methodenaufrufs vom Empfängerobjekt folgt, dass nicht immer schon zur Übersetzungszeit entschieden werden kann, welche Methodenimplementierung bei einem Methodenaufruf ausgewählt werden muss. Wenn nämlich das Empfängerobjekt durch eine Variable benannt oder von einem Ausdruck geliefert wird, kann die Zuordnung einer Methodendefinition zu einem Nachrichtenausdruck erst zum Zeitpunkt der Auswertung des Nachrichtenausdrucks und damit erst zur Laufzeit erfolgen. Man nennt diesen Vorgang **dynamisches Binden** (im Gegensatz zum **statischen Binden**, bei dem ein Aufruf schon zur Übersetzungszeit an eine Implementierung gebunden wird); es handelt sich dabei um eine von den nur zwei *primitiven Kontrollstrukturen* SMALLTALKS (s. Abschnitt 4.5.2).

dynamisches Binden

Das dynamische Binden ist eine der charakteristischen Eigenschaften der objektorientierten Programmierung. Sie wird auch als **Polymorphismus**

zentrale Bedeutung des dynamischen Bindens



oder **Polymorphie**²² bezeichnet. Auf die Details des dynamischen Bindens können wir erst in der nächsten Kurseinheit (Kapitel 12) zu sprechen kommen und auf Polymorphie erst in Kapitel 26, da uns hier noch zu viel fehlt. Wir vermerken aber schon jetzt, dass es sich dabei um eine *versteckte Fallunterscheidung* handelt: Ein und derselbe Methodenaufruf kann fallweise unterschiedliche Methoden aufrufen (bzw. deren Ausführung veranlassen). Auf diese Eigenschaft kommen wir schon in den Abschnitten 4.5 und 4.6 zurück.

Steht einmal fest, welche (Implementierung einer) Methode aufgerufen wird, erfolgt als nächstes die Versorgung der formalen Parameter mit Objekten. Zu diesem Zweck findet mit dem Aufruf eine **implizite Zuweisung** (also eine ohne Vorkommen des Zuweisungsoperators im Programmtext) der *tatsächlichen Parameter* des Aufrufs an die formalen Parameter der Methode statt. Tatsächliche Parameter sind dabei die Objekte, die an den Positionen der formalen Parameter der aufgerufenen Methode stehen. Manchmal werden auch die Variablen, die an den entsprechenden Stellen beim Methodenaufruf stehen, als tatsächliche Parameter bezeichnet, aber erstens müssen dort nicht unbedingt Variablen, sondern können beliebige Ausdrücke stehen und zweitens können diese Variablen selbst formale Parameter sein, nämlich die der Methode, die den Methodenaufruf enthält. So ist in der Methodendefinition

implizite Zuweisungen der tatsächlichen an die formalen Parameter

```
106 m: a  
107   self n: a
```

a der formale Parameter von m:. Der tatsächliche Parameter von m: ist ein Objekt, das beim Aufruf von m: genannt wird (hier nicht zu sehen). Dieses Objekt ist dann auch tatsächlicher Parameter des Aufrufs von n:, da dieser von a, dem formalen Parameter von m:, geliefert wird.

Je nach Sichtweise erfolgt bei Ausführung der Return-Anweisung eine weitere implizite Zuweisung, nämlich die des Objekts, zu dem der Ausdruck der Return-Anweisung auswertet, an die „Variable“ Methodename (der ja an der Stelle des Methodenaufrufs ähnlich wie eine Variable für ein Objekt steht). Dies ist vor allem im Zusammenhang mit der Typprüfung, die es jedoch in SMALLTALK nicht gibt (s. Kurseinheit 3), eine wichtige Vorstellung. An der Aufrufstelle selbst steht dann häufig noch eine *explizite Zuweisung*, nämlich wenn das Ergebnis des Aufrufs (das Rückgabebewert) einer Variable zugewiesen werden soll.

implizite Zuweisung des Rückgabewertes

Selbsttestaufgabe 4.1

Benennen Sie die expliziten und impliziten Zuweisungen für

²² Es ist mir bis heute nicht klar, wann man von *Polymorphie* und wann von *Polymorphismus* spricht. Im Gebrauch scheint sich anzudeuten, dass man von Inklusionspolymorphie und von parametrischem Polymorphismus spricht, aber einen Grund dafür kann ich nicht erkennen.



mit der Methode

```
109 f: g
110 ^ 2
```

In Abschnitt 1.5 waren wir auf den Unterschied zwischen Wertsemantik und Verweissemantik eingegangen. Damit verwandt (und ähnlich benannt) ist die Unterscheidung von **Call by reference** und **Call by value** beim Methodenaufruf: Beim Call by value wird dem formalen Parameter der tatsächliche Parameter als Wert zugewiesen, beim Call by reference hingegen nur eine Referenz. Diese Referenz ist jedoch nicht, wie man vielleicht glauben könnte, eine auf den tatsächlichen Parameter als Objekt, sondern eine auf den tatsächlichen Parameter als Variable (s. o.), der dazu aber eben auch eine Variable sein muss. Dies hat zur Folge, dass Zuweisungen zum formalen Parameter in der Methode unter Call by reference auch die Variable, die den tatsächlichen Parameter darstellt, betreffen — die beiden sind gewissermaßen eins. Mit Call by reference ist es also möglich, dass eine Methode auch über ihre tatsächlichen Parameter, wenn sie denn Variablen sind, Objekte zurückgibt — sie werden damit zu Ein- und Ausgabeparametern der Methode. Bei Call by value bleibt die Zuweisung an die formalen Parameter jedoch ohne Bedeutung für die tatsächlichen — sie sind also reine Eingabeparameter.

**Call by reference und
Call by value**

Nun werden Sie vielleicht einwenden, dass in SMALLTALK formale Parameter Pseudovariablen sind und deswegen gar keine Zuweisung an sie erlaubt ist (außer der *impliziten Zuweisung* beim Aufruf). Das ist richtig. Tatsächlich gibt es in SMALLTALK ein Call by reference auch gar nicht (in JAVA übrigens auch nicht). Gleichzeitig haben aber in SMALLTALK Variablen grundsätzlich Referenzsemantik, so dass bei der Zuweisung der tatsächlichen an die formalen Parameter keine Objekte, sondern lediglich Verweise an diese übergeben werden. Wenn man dann innerhalb der Methode etwas an diesen Objekten ändert (ihren Zustand), dann betrifft das immer die tatsächlichen Parameterobjekte und somit auch den „Inhalt“ der tatsächlichen Parametervariablen (wenn es denn Variablen und keine anderen Ausdrücke sind und wenn sie wirklich ein Objekt zum Inhalt hätten; vgl. Abschnitt 1.5): Es sind schließlich dieselben Objekte. Insbesondere erhalten die Methoden also keine Kopien dieser Objekte, sondern lediglich Kopien der Referenzen. Um den Inhalt von tatsächlichen Parametervariablen zu ändern (also sie auf andere Objekte zeigen zu lassen), bräuchte man auch in SMALLTALK (und JAVA) Call by reference, was es dort aber nicht gibt. Es stellt dies eine echte Beschränkung der Programmierung dar; sie wurde denn auch in C# aufgehoben.

**Unterschied zu
Referenz- und
Wertsemantik;
SMALLTALK kennt nur
Call by value**

Bleibt noch die Frage, was passiert, wenn ein Methodenaufruf ins Leere läuft. Da in SMALLTALK Ausdrücke beliebige Objekte liefern können, kann der Compiler für einen Nachrichtenausdruck nicht garantieren, dass das Empfängerobjekt auch über eine entsprechende Methode verfügt. Da der Nachrichtenausdruck in einen dynamisch gebundenen Methodenaufruf übersetzt wird, dessen Ausführung direkt von der virtuellen Maschine

**Aufruf nicht
vorhandener
Methoden**

SMALLTALKS



vorgenommen wird, ist die Frage, wie das Programm sinnvoll mit einem solchen Laufzeitfehler umgehen soll. Tatsächlich passiert in etwa folgendes: Die virtuelle Maschine macht aus dem Methodenaufruf einen Nachrichtenselektor (und zwar den, aus dem er bei der Übersetzung hervorgegangen ist) und sendet diesen als Parameter an eine vorgegebene Methode `doesNotUnderstand`: des ursprünglichen Empfängers. Diese reagiert typischerweise mit einer der Ausgabe einer Fehlermeldung `<Objekt> does not understand: <Nachrichtenselektor>`; sie kann aber geändert werden, um anders als standardmäßig vorgesehen auf den Fehler zu reagieren.

4.3.3 Methoden als Parameter

Nun wurde eingangs dieses Kapitels vom Mysterium des Nachrichtenverands gesprochen. Tatsächlich ist aber zumindest in SMALLTALK seine Realisierung in Form von Methodenaufrufen nur ein Zugeständnis an die Ausführungseffizienz. Und so sind denn auch in SMALLTALK Nachrichten (oder vielmehr Nachrichtenselektoren) auch Objekte — schließlich soll dort ja alles ein Objekt sein. Um tatsächlich als Nachrichtenobjekt an ein Objekt verschickt zu werden, muss man sich aber einer speziellen Methode (genauer: eines speziellen Methodenaufrufs), `perform:`, bedienen, der es erlaubt, einem Empfängerobjekt eine Nachricht als Objekt (wenn auch nur als Parameter von `perform:`) zu senden. Das Empfängerobjekt reagiert darauf mit der Abarbeitung der zur Nachricht passenden Methode ganz so, als hätte es direkt einen entsprechenden Methodenaufruf erhalten. So wertet zum Beispiel

... und es gibt ihn
doch, den
Nachrichtenversand

```
111 nil perform: #isNil
```

genau wie

```
112 nil isNil
```

zu `true` aus. Der Nachrichtenselektor ist immer ein Symbol (`#isNil` im gegebenen Beispiel) und darf beim „Versand“ mittels `perform:`, anders als beim direkten Aufruf, auch in einer Variable gespeichert sein. Bei binären und höherstelligen Nachrichten braucht man zusätzlich noch die Argumente (Parameter) zum Nachrichtenselektor; sie können durch Erweiterung von `perform:` zu `perform:with:`, `perform:with:with:` usw. angehängt werden. So ist dann beispielsweise der Ausdruck

```
113 1 perform: #+ with: 2
```

äquivalent zu `1 + 2`.



4.3.4 Verbergen der Repräsentation des Zustands hinter Methoden

In SMALLTALK sind die Instanzvariablen eines Objekts nur für das Objekt selbst sichtbar (und damit auch zugreifbar).²³ Genauer sind seine Methoden die einzigen Stellen im ganzen Programm, an denen auf die Instanzvariablen des Objekts, dem die Methoden zugeordnet sind, direkt zugegriffen werden kann. Die Struktur des Objekts bleibt somit verborgen (das *Implementationsgeheimnis*) und sein *Zustand* wird *gekapselt* (s. Abschnitt 3.2 sowie unten für ein praktisches Beispiel).

Um die Belegung der Instanzvariablen und damit den Zustand eines Objektes auszulesen oder verändern zu können, sind also Methoden notwendig. Um beispielsweise den Wert einer Instanzvariable mit Namen **a** auszulesen, muss das Objekt eine Methode vorsehen, die den Wert von **a** zurückgibt. Diese (parameterlose) Methode wird in SMALLTALK üblicherweise wie folgt notiert:

**Zugriffsmethoden,
Getter und Setter**

```
114 a  
115     "gib den Wert von a zurück"  
116     ^ a
```

Sie entspricht im wesentlichen einem auch in JAVA gebräuchlichen sog. **Getter** einer ansonsten nicht zugreifbaren Variable. Die Namensgleichheit von Methode und Variable soll dabei nicht darüber hinwegtäuschen, dass es sich bei beiden um verschiedene Dinge handelt — Methode und Variable könnten genauso gut verschieden benannt werden.

Um den Wert von **a** zu setzen, definiert man üblicherweise die folgende, auch **Setter** genannte Methode:

```
117 a: einWert  
118     "setze den Wert von a auf einWert"  
119     a := einWert
```

Getter und Setter werden zusammen auch als **Zugriffsmethoden** (oder **Accessoren**; engl. accessor methods) bezeichnet.

Die obigen Zugriffsmethoden werden aufgerufen, indem man dem Objekt, zu dem die Instanzvariable **a** gehört, die Nachricht **a** (zum Lesen) bzw. **a:** mit einem Objekt als Parameter (zum Schreiben) schickt. Das Empfängerobjekt antwortet darauf im ersten Fall mit Rückgabe von **a** und im zweiten Fall mit Rückgabe von sich selbst. Man beachte, dass bei der Rückgabe keine Kopie, sondern der Inhalt der Variable (bzw. ein Verweis darauf) zurückgegeben wird. Nach Auswertung des Ausdrucks

²³ Dass dies in anderen Sprachen anders ist (z. B. in JAVA), kann man durchaus als Entwurfsfehler ansehen. Auf der anderen Seite schützt die mangelnde Sichtbarkeit von Instanzvariablen ja nicht vor dem Zugriff auf Objekte, auf die sie verweisen (wegen des möglichen *Aliasing*), so dass man sich an dieser Stelle nicht versteigen sollte.



```
120 b := einObjekt a
```

verweisen also **b** und die Instanzvariable **a** von **einObjekt** (genauer: dem von **einObjekt** bezeichneten Objekt) auf dasselbe Objekt, genauso wie nach Auswertung von

```
121 einObjekt a: b
```

Bei beiden ist hinterher **b** ein Alias auf **a**, was de facto die Kapselung des Zustandes von **einObjekt** durchbricht.

Es soll nochmals darauf hingewiesen werden, dass im ersten Ausdruck **a** eine Nachricht darstellt und keinesfalls der Name der Instanzvariable des Objekts ist. Insbesondere handelt es sich bei dem (Teil-)Ausdruck **einObjekt a** mitnichten um das Äquivalent zu dem aus JAVA bekannten **einObjekt.a**, sondern vielmehr dem von **einObjekt.getA()** (wobei man die Methode **getA()** in JAVA per Konvention natürlich genauso gut nur **a()** nennen könnte). Ein direkter Zugriff auf die Instanzvariable von außen wie in JAVA ist in SMALLTALK nicht möglich. Man kann also den Zugriff auf eine Instanzvariable verhindern, indem man einfach keine Zugriffsmethoden dafür vorsieht; man kann ihn auf *Nur-Lesen* oder *Nur-Schreiben* beschränken, indem man nur die jeweilige Zugriffsmethode zur Verfügung stellt.

Man beachte schließlich, dass anders als benannte Instanzvariablen indizierte Instanzvariablen auch von dem Objekt, zu dem sie gehören, nicht direkt, sondern nur über die beiden vordefinierten Nachrichten **at:** und **at:put:** gelesen und geschrieben werden können:

**Zugriffsmethoden für
indizierte
Instanzvariablen**

```
122 einObjekt at: i
```

liefert den Wert der Instanzvariable mit dem Index **i**,

```
123 einObjekt at: i put: einAnderesObjekt
```

setzt ihn. Es ist in SMALLTALK also nicht möglich, indizierte Instanzvariablen eines Objekts (im Gegensatz zu benannten) durch Nicht-Deklaration von Zugriffsmethoden zu verbergen (vor Zugriff zu schützen). Zugleich folgt aus der festen Vorgabe der beiden Zugriffsmethoden, dass jedes Objekt nur genau eine (unbenannte) Menge von indizierten Instanzvariablen haben kann.

Die ausschließliche Abfrage und Änderung des Zustandes von Objekten über Methoden hat den Vorteil, dass man sich nicht festlegt, wie man den Zustand eines Objekts tatsächlich codiert. So kann man beispielsweise einem Punktobjekt Methoden zum Setzen und Abfragen sowohl von kartesischen als auch von polaren Koordinaten zuordnen, muss aber nur Instanzvariablen für eine Art von Koordinaten vorsehen und kann die anderen jeweils berechnen:

**Repräsentation des
Zustands als
Implementations-
geheimnis**

```
124 kartesischX: a y: b
125     "setzt kartesische Werte"
126     x := a.
127     y := b.
```



```

128 x
129     "liefert die x-Koordinate"
130     ^ x

131 y
132     "liefert die y-Koordinate"
133     ^ y

134 polarRadius: r winkel: a
135     "setzt polare Werte"
136     x := a cos * r.
137     y := a sin * r

138 radius
139     "liefert die Radius-Koordinate"
140     ^ (x * x + (y * y)) sqrt

141 winkel
142     "liefert die Winkel-Koordinate"
143     ^ (x / self radius) arcCos

```

Man könnte die Koordinaten natürlich genauso gut in polarer Form speichern und die kartesischen berechnen — für die Benutzerin dieser Objekte spielt das keine Rolle. Man betrachtet die Art und Weise, wie ein Objekt seinen Zustand codiert, als sein *Implementationsgeheimnis* und die Menge der Methodensignaturen, die den Zugriff auf das Objekt (seinen Zustand) erlauben (das *Protokoll*), als sein *Interface*. Mehr dazu in Abschnitt 4.3.8.

**Implementations-
geheimnis und
Interface**

4.3.5 Operationen auf zustandslosen (unveränderlichen) Objekten

Auf unveränderlichen Objekten ausgeführte Methoden oder Operationen können deren Zustand naturgemäß nicht ändern. So ändert beispielsweise die Addition von Zahlen nichts an ihren Operanden, sondern liefert als Ergebnis ein anderes Objekt. Derartige Methoden (oder Operationen) sind also klassische Funktionen: Sie berechnen anhand eines oder mehrerer Argumente (wovon eines das Empfängerobjekt ist) ein Ergebnis, und das ohne *Seiteneffekte* wie die Zustandsänderungen der Argumente.

Bei anderen Objekten wie z. B. den Punktobjekten des Abschnitts 4.3.4 ist es hingegen fraglich, ob Operationen wie die Addition den Zustand des Empfängers verändern oder ein neues Objekt zurückgeben sollen: Bei einer Veränderung des Empfängers würde die Addition dann eher einer Translation gleichkommen (weswegen die entsprechende Methode auch so genannt werden sollte). Um jedoch zu zeigen, wie man neue Objekte erzeugen und zurückgeben kann, fehlt uns hier noch einiges; wir kommen erst in Abschnitt 7.3 darauf zurück.



4.3.6 Konstante Methoden

In SMALLTALK gibt es keine frei bezeichbaren *Konstanten*²⁴, sondern nur *Literale* (s. Abschnitt 1.2). Da verschiedene Vorkommen gleicher Literale aber (außer bei Symbolen) verschiedene Objekte erzeugen, sind Literale nicht für alle Zwecke ausreichend. Es gibt dafür einen Trick, mit dem man dasselbe Literal mehrfach verwenden kann, nämlich durch eine sog. **konstante Methode**.

```
144 pi
145     "na ja, so ungefähr ..."
146     ^ 3.142
```

ist eine solche konstante Methode. An ihr ist nichts weiter konstant, als dass sie immer dasselbe Objekt zurückgibt. Der Trick besteht nämlich darin, dass das zum Literal 3.142 gehörende Zahlobjekt nur einmal, nämlich zur Übersetzungszeit der Methode, erzeugt wird und die Ausführung der Methode immer auf dieses und damit dasselbe Objekt zurückgreift.

Nun gibt es leider zwei Probleme. Das erste ist offensichtlich, dass bei einer erneuten Übersetzung der Methode auch ein neues Objekt erzeugt wird, das dann mit früher zurückgegebenen nicht mehr identisch ist (von den bereits bekannten Ausnahmen abgesehen). Das ist immer dann ein Problem, wenn das früher zurückgegebene Objekt in Variablen gespeichert wurde und nun mit dem neuen auf Identität verglichen werden soll.²⁵ Das sollte man also tunlichst unterlassen.

Probleme konstanter Methoden

Das zweite Problem ist noch etwas subtiler: Bei der konstanten Methode

```
147 einsZweiDrei
148     ^(1 2 3)
```

eines Objekts o und nach Auswertung der Anweisungen

```
149 a := o einsZweiDrei
150 a at: 1 put: 0
151 b := o einsZweiDrei
```

enthält b an erster Stelle ebenfalls eine 0! Die vermeintlich konstante Methode ist also alles andere als konstant!! Interessanterweise ist das Ergebnis dieses Experiments genau konvers

²⁴ Für gewöhnlich sind Konstanten Namen für (unveränderliche Werte); man kann Sie also als Variablen auffassen, denen nur einmal (bei der Initialisierung) ein Wert zugewiesen wird. Die Definition von Konstanten erfordert dann aber ein Schlüsselwort, um sie von anderen Variablen zu unterscheiden.

²⁵ Man bedenke, dass in SMALLTALK das (neu) Komplizieren einer Methode nicht bedeutet, dass das Programm danach neu gestartet werden muss — das Programm läuft vielmehr weiter und alle Objekte mit ihren Variablenbelegungen bleiben erhalten.



zum ersten Problem: Die Identität der von der konstanten Methode zurückgegebenen Objekte bleibt erhalten, ihre Erscheinung ändert sich aber (beim ersten Problem blieb die Erscheinung gleich, aber die Identität änderte sich).

Zumindest das zweite Problem lässt sich eindämmen, indem man auf Sprachebene durch Literale erzeugte Objekte als unveränderlich annimmt und Änderungen der Art von Zeile 150 entsprechend verbietet. Dies machen einige SMALLTALK-Dialekte auch tatsächlich.²⁶ Die eigentliche Erkenntnis ist aber, dass die Referenzsemantik von Variablen und das damit verbundene Aliasing von Objekten zu höchst subtilen Problemen führen kann, derer man sich immer gewahr sein sollte.

4.3.7 Primitive Methoden

Zwar ist SMALLTALK über weite Teile in sich selbst definiert (was sich darin äußert, dass praktisch die gesamte Sprachdefinition in Form von Methoden hinterlegt und damit für die Programmiererin nicht nur sichtbar, sondern auch änderbar ist), aber für einige primitive Operationen greift es doch auf native Implementierungen zurück. Dazu zählt z.B. die Addition (+) für kleine Integer oder auch der Zugriff auf indizierte Variablen mittels `at:` und `at:put:.` Die entsprechenden Methoden sind in SQUEAK wie folgt implementiert:

```
152 + aNumber
153   <primitive: 1>
154   ^ super + aNumber

155 at: index
156   <primitive: 60>
157   index isInteger ifTrue:
158     [self class isVariable
159       ifTrue: [self errorSubscriptBounds: index]
160       ifFalse: [self errorNotIndexable]].
161   index isNumber
162     ifTrue: [^self at: index asInteger]
163     ifFalse: [self errorNonIntegerIndex]

164 at: index put: value
165   <primitive: 61>
166   index isInteger ifTrue:
167     [self class isVariable
168       ifTrue: [(index >= 1 and: [index <= self size])
169         ifTrue: [self errorImproperStore]
170         ifFalse: [self errorSubscriptBounds: index]]
171       ifFalse: [self errorNotIndexable]].
172   index isNumber
173     ifTrue: [^self at: index asInteger put: value]
174     ifFalse: [self errorNonIntegerIndex]
```

²⁶ In JAVA sind Array-Literale übrigens nur in New-Ausdrücken erlaubt; die dadurch beschriebenen Array-Objekte werden erst zur Laufzeit und dann immer wieder neu erzeugt, so dass sich keine Aliase ergeben.



Dabei stehen die in spitzen Klammern stehenden Ausdrücke jeweils für Aufrufe von primitiven Methoden, die, da man sie als Programmiererin nicht selbst verwenden soll, nur durchnummiert wurden. Die Anweisungen nach den Aufrufen der primitiven Methoden werden nur ausgeführt, wenn die primitive Methode nicht erfolgreich war. Das bedingt, dass aus einer primitiven Methode mittels Return direkt zurückgesprungen werden kann, und zwar dorthin, wo die Methode `+`, `at:` bzw. `at:put:` aufgerufen wurde. Dieses Verhalten, das einigermaßen ungewöhnlich erscheint, wird uns im Kontext von Blöcken (Abschnitt 4.4) wieder begegnen.

4.3.8 Protokoll

In SMALLTALK ist das **Interface** eines Objekts die Menge der Nachrichten, die es versteht. Dieses Interface wird in Form der sog. **Protokollbeschreibung** o. kurz des **Protokolls** spezifiziert, die aus den *Methodensignaturen* und den *Kommentaren* der zu den Nachrichten passenden Methoden besteht und der die **Implementation**, also die Liste der *Methodenrumpfe*, gegenübersteht. Letztere sind, zusammen mit den Instanzvariablen, das **Implementationsgeheimnis** eines Objekts, das hinter seiner Protokollbeschreibung (dem Interface) verborgen wird. Die Kommentare dürfen übrigens, wie in den meisten anderen Sprachen auch, als (schwacher) Ersatz für eine formale Spezifikation des Verhaltens, das in einer Methode implementiert wird, angesehen werden (vgl. dazu Abschnitt 52.6 in Kurseinheit 5).

Um die Programmierung zu erleichtern, wird in den meisten SMALLTALK-Systemen das Protokoll von Objekten in sog. **Nachrichtenkategorien** eingeteilt, die jeweils einen Namen tragen, der die in der Kategorie enthaltenen Namen zusammenfasst. Da jede Methode in genau eine Nachrichtenkategorie fallen muss, stellen diese eine Partitionierung des Interfaces eines Objekts dar. Unter den Kategorien sind solche, die das Wort „private“ enthalten; deren Methoden sollten dann nicht „von außerhalb“, also nur vom Objekt selbst (über `self`) aufgerufen werden. Dies wird jedoch nicht vom Compiler erzwungen. Nachrichtenkategorien haben auch sonst keinerlei die Programmausführung betreffende Bedeutung, sondern dienen lediglich der besseren Lesbarkeit.

Einteilung in
Nachrichten-
kategorien

Wie Sie in der nächsten Kurseinheit lernen werden, werden Protokolle in SMALLTALK nicht auf Objektebene, sondern auf Klassenebene spezifiziert. In STRONGTALK, einer Erweiterung von SMALLTALK um ein (optionales) Typsystem, werden Protokolle dann zu Typen erhoben (s. Kurseinheit 3, Kapitel 20). Da Protokolle nur Methoden enthalten, sind sie den Interfaces JAVAs sehr ähnlich. Tatsächlich werden Protokolle in STRONGTALK auch manchmal Interfaces genannt.

4.4 Blöcke

Wir kommen nun zu einer der wichtigsten Ausprägungen von SMALLTALKs Alles-ist-ein-Objekt-Motto: den **Blöcken**. Genau wie eine Methode ist ein



Block eine abgegrenzte Sequenz, oder Folge, von Anweisungen. Anders als eine Methode ist ein Block jedoch nicht benannt; er kann aber benannt werden, indem er einer Variable zugewiesen wird.

Um auszudrücken, dass eine Sequenz von Ausdrücken ein Block ist, wird die Sequenz mit eckigen Klammern markiert. So ist

Definition von Blöcken

```
175 [ | temp | temp := x. x := y. y := temp ]
```

die Definition eines Blocks, der aus der Deklaration der Variable `temp` und drei Zuweisungen besteht. Die Variablen `x` und `y` seien dabei außerhalb des Blocks, im **Kontext des Blocks**, deklariert. Dabei ist der Kontext des Blocks die Methode, in der er definiert wurde.²⁷

Bei der Ausführung des obigen Blocks wird ein neues Blockobjekt erzeugt. Mittels

Kontext eines Blocks

Ausführung von Blockausdrücken

```
176 swap := [ | temp | temp := x. x := y. y := temp ]
```

wird der Block einer Variable `swap` zugewiesen. Die Anweisungen, die den Block ausmachen, werden dabei *nicht* ausgeführt, selbst dann nicht, wenn der Block (wie in Zeile 175) isoliert steht und ausgeführt wird (das dabei erzeugte Objekt bleibt namenlos und wird von der Speicherbereinigung wieder entfernt).

Um die Anweisungen, die einen Block ausmachen, zur Ausführung zu bringen, muss man ihn auswerten. Dazu schickt man ihm die Nachricht `value`. Der Ausdruck

Auswertung von Blöcken

```
177 swap value
```

bewirkt, dass die Variablen `x` und `y` aus dem Kontext des Blocks ihren Wert tauschen. Das Objekt, zu dem `swap value` auswertet, ist das Objekt, zu dem die letzte Anweisung auswertet (s. Abschnitt 4.2; im obigen Beispiel also der Inhalt von `temp`, der derselbe ist wie der von `x` aus dem Kontext).

Rückgabewert der Methode `value` ist zunächst immer das Objekt, zu dem der letzte Ausdruck eines Blocks auswertet, im obigen Beispiel das durch `temp` benannte Objekt.

Wert eines Blocks

4.4.1 Home context und Closure

Da Blöcke Objekte sind, die Variablen zugewiesen werden können, können sie auch an andere Methoden übergeben werden. Werden sie dort (mittels `value`) ausgewertet, dann

²⁷ In einem Workspace definierte und mittels `doIt` usw. ausgeführte Blöcke werden zu diesem Zweck in (temporäre) Methoden übersetzt, die keinem Objekt (oder `nil`) zugeordnet sind.



findet die Auswertung in einem anderen Kontext statt. In diesem sind die „freien“ Variablen des Blocks (also die, die nicht selbst als lokale Variablen deklariert wurden; x und y in Zeile 175) aber gar nicht zugreifbar. Der Block nimmt deswegen seinen Kontext mit (oder, richtiger, der Kontext ist im Block miteingeschlossen). Der Kontext, in dem ein Block definiert wurde (in dem das ihn repräsentierende Objekt erzeugt wurde), nennt man seinen **Home context**. Die Auswertung eines Blocks erfolgt stets in seinem Home context, insbesondere auch dann, wenn ihm **value** in einem anderen Kontext gesendet wurde.

Das folgende Beispiel zweier Methoden soll den Sachverhalt erläutern:

```
178 homeContext
179   | x |
180   x := 2.
181   self otherContext: [Transcript show: x printString]
182 otherContext: x
183   x value
```

gibt auf dem Transcript 2 aus, obwohl im Kontext von **otherContext**: die Variable x einen Block und nicht 2 zum Wert hat. Die (Werte der) *Pseudovariablen* **self** und **super** zählen übrigens auch zum Home context; dies ist vor allem im Zusammenhang mit dem dynamischen Binden (Kapitel 12 in Kurseinheit 2) interessant.

Dass ein Block aus seinem Home context herausgelöst und in einem anderen gespeichert werden kann beinhaltet das Problem, dass die lokale Variablen des Home contexts schon verschwunden sein können, wenn der Block ausgewertet wird. Die durch den Block „eingefangenen“ lokalen Variablen (inkl. formale Parameter) müssen daher unabhängig von der Ausführung der Methoden, die sie definieren, weiterleben. Die Umsetzung von Blöcken durch den SMALLTALK-Compiler ist alles andere als trivial und verschiedene SMALLTALK-Systeme unterscheiden sich darin zum Teil erheblich voneinander, was sich (leider) auch in unterschiedlichem Verhalten äußert.

Die Blöcke SMALLTALKS heißen in anderen Sprachen übrigens (lexikalische) **Closures**; sie werden für die sog. *Lambda-Ausdrücke*, also (anonyme)

Blöcke in anderen Sprachen

Funktionen, die selbst Objekte oder Werte sind und die deswegen aus ihrem Kontext herausgelöst und in andere verschoben werden können, gebraucht. Dabei unterscheiden sich die Sprachen zum Teil erheblich darin, was alles in eine Closure einbezogen werden kann; so können die lokalen Namen (Variablen) beispielsweise auf Konstanten eingeschränkt werden, um zu vermeiden, dass *temporäre Variablen* weiterleben müssen, weil sie in einer Closure enthalten sind.

4.4.2 Continuation

Das Konzept des Home context eines Blocks geht aber noch weiter: Es gehören nicht nur die sichtbaren Variablen aus dem Kontext der Definition des Blocks dazu, sondern auch der sog. *Call stack*, also der Speicher, in dem die Rücksprungadressen von Methodenaufrufen



abgelegt werden. Entsprechend bedeutet eine *Return-Anweisung* innerhalb eines Blocks bei seiner Auswertung auch stets die sofortige Rückkehr in den den Block definierenden *Home context* (der Methode) und nicht in den Kontext (in der Methode), in dem (in der) der Block, durch Senden von **value**, ausgewertet wird. Das nachfolgende Beispiel zeigt das:

```
184 homeContext
185   self otherContext: [ ^ Transcript show: 'home' ]
186 otherContext: x
187   x value.
188   Transcript show: 'other'
```

Der Aufruf der Methode **homeContext** gibt „home“, aber nicht auch noch „other“ auf der Konsole aus. Man nennt dieses Konzept, das ebenfalls aus der Welt der *funktionalen Programmierung* stammt, auch **Continuation**. Continuations spielen bei der Implementierung von Kontrollstrukturen in SMALLTALK eine entscheidende Rolle (s. Kapitel 4.6).

Das Prinzip der Continuation gilt übrigens auch für geschachtelte Blöcke:

```
189 [[ ^ true] value. [ ^ false] value] value
```

liefert **true**, nicht **false**. Return-Anweisung verlässt auch alle umschließenden Blöcke, und zwar sofort. Dies liegt daran, dass explizite Return-Anweisungen aus Blöcken immer dahin zurückkehren, von wo die Definition des Blocks angestoßen wurde.

Continuations können zu Laufzeitfehlern führen, nämlich wenn durch sie von einer Methode zurückgekehrt werden soll, die bereits beendet wurde. Wenn beispielsweise das Objekt **o** über die Methode

Laufzeitfehler bei Continuations

```
190 merkwuerdig
191   ^ [ ^ self]
```

verfügt, dann führt

```
192 o merkwuerdig value
```

zu einem Laufzeitfehler, da die Auswertung des von **merkwuerdig** mit ihrer Beendigung zurückgegebenen Blocks **merkwuerdig** noch einmal beenden müsste, was aber nicht geht. Return-Anweisungen in Blöcken sind u. a. deswegen ein umstrittenes Konzept. Für SMALLTALK sind sie aber unverzichtbar, da mit ihnen fast alle *Kontrollstrukturen* implementiert werden (s. Kapitel 4.6).

4.4.3 Parametrisierte Blöcke

Die Blöcke von SMALLTALK können auch als anonyme Funktionen aufgefasst werden, wobei alle bisherigen parameterlos waren: Der Bezug zu ihrer Umwelt wurde ausschließlich über den Home context hergestellt. Es aber auch möglich, Blöcke mit Parametern zu versehen,



die bei ihrer Auswertung an Objekte aus dem Kontext der Auswertung gebunden werden können. Den obigen Swap-Block könnte man also auch wie folgt definieren wollen:

```
193 swap := [ :x :y | | temp | temp := x. x := y. y := temp ]
```

wobei die jeweils von einem Doppelpunkt eingeleiteten und vom Rest des Blocks durch einen senkrechten Strich abgetrennten Variablen **x** und **y** die *formalen Parameter* des Blocks sind und das — analog zu einer Methode — in Strichen eingeschlossene **temp** eine lokale Variable ist. Der Variable **swap** wie oben zugewiesen kann der Block mittels

```
194 swap value: a value: b
```

ausgewertet werden, wenn **a** und **b** Variablen aus dem Kontext der Auswertung sind.

Selbsttestaufgabe 4.2

- (a) Versuchen Sie, durch Überlegen oder Ausprobieren herauszufinden, ob die Auswertung des Blocks **swap** aus Zeile 193 seinen (mutmaßlichen) Zweck erfüllt, also beispielsweise in Zeile 194 den Wert der Variablen **a** und **b** tauscht.
- (b) Wie sieht das beim Parameterlosen **swap** aus Zeile 176 (in einem geeigneten Kontext) aus?
- (c) Begründen Sie, warum es sinnvoll ist, die explizite Zuweisung an die formalen Parameter eines Blocks (im obigen Beispiel **x** und **y**) innerhalb des Blocks zu verbieten.

4.5 Kontrollstrukturen

Kontrollstrukturen regeln den Ablauf des Programms, also die Reihenfolge der Schritte, aus denen seine Ausführung besteht. Anders als in anderen Programmiersprachen gibt es in SMALLTALK nur zwei Kontrollstrukturen, nämlich die Sequenz und den dynamisch gebundenen Methodenaufruf; alle anderen, inklusive der Verzweigung und der Wiederholung (Schleife), müssen durch diese simuliert werden. Dies ist möglich, weil SMALLTALK Blöcke hat und weil in SMALLTALK (so wie in allen anderen objektorientierten Programmiersprachen) der Methodenaufruf variabler ausfällt als der gewöhnliche Prozedur- oder Funktionsaufruf, wie Sie ihn vielleicht von Sprachen wie PASCAL oder C her kennen: Er enthält, wie bereits in Abschnitt 4.3.2 angedeutet, eine versteckte Fallunterscheidung in Form des *dynamischen Bindens*.

4.5.1 Sequenz

Die Sequenz als Kontrollstruktur besagt lediglich, dass textuell aufeinanderfolgende Anweisungen eines Programms (einer Methode) auch zeitlich nacheinander ausgeführt werden. Die zeitliche Sequenz aufeinander folgender Anweisungen kann lediglich durch andere Kontrollstrukturen (in SMALLTALK nur durch den Methodenaufruf; s. u.) unterbrochen werden.



Dies gilt auch für *parallele Ausführung*, die man sich wie die gleichzeitige Abarbeitung zweier sequentieller Programme auf denselben Objekten vorstellen kann (s. Kapitel 16).

4.5.2 Dynamisch gebundener Methodenaufruf

Wie bereits in Abschnitt 4.3.2 erläutert, verbirgt sich hinter dem Nachrichtenversand in SMALLTALK der Methodenaufruf. Wann immer ein Objekt eine Nachricht an ein Empfängerobjekt verschickt, wechselt der Kontrollfluss damit zum Empfängerobjekt, genauer zu der Methode des Empfängerobjekts, das zur Reaktion auf die Nachricht vorgesehen ist. Nach der Abarbeitung der Methode kehrt der Kontrollfluss an das sendende Objekt (genauer: zu der Methode, aus der die Nachricht versandt wurde) zurück und setzt seine Arbeit dort fort. Bei der Rückkehr wird auch das Ergebnis der Methode, (eine Referenz auf) ein Objekt, geliefert, das dann an der Stelle des Nachrichtenausdrucks, der den Methodenaufruf bewirkt hat, eingesetzt wird. Der genaue Mechanismus des dynamischen Bindens in SMALLTALK wird in Kapitel 12 von Kurseinheit 2 untersucht.

4.6 Abgeleitete Kontrollstrukturen

Wenn Sie schon in anderen Programmiersprachen wie z. B. PASCAL, C oder JAVA programmiert haben, dann kennen Sie sicher *Schlüsselwörter* wie *if*, *else*, *for* und *while*. Diese Schlüsselwörter stehen für Kontrollstrukturen, feste Bestandteile der Sprache, die für die Steuerung des Ablaufs eines Programms durch die Programmiererin vorgesehen sind. Man hat sich irgendwann einmal (im Zuge der Diskussion zur sog. *strukturierten Programmierung*) darauf festgelegt, dass jede Programmiersprache über die Kontrollstrukturen Sequenz, Verzweigung, Wiederholung (*Iteration*) und Aufruf verfügen sollte. Während die einfache Sequenz von Anweisungen während der Ausführung durch die lineare Folge der Anweisungen im Programmtext vorgegeben ist, sind für alle Abweichungen vom linearen Kontrollfluss, also für Verzweigung, Wiederholung und Aufruf, spezielle Flusssteuerungskonstrukte vorgesehen. Das *Goto* gehört übrigens nicht dazu; es gilt seit dem Aufkommen der strukturierten Programmierung als verpönt.



WIKIPEDIA



In SMALLTALK hat man die durch die Syntax der Sprache vorgesehenen Kontrollstrukturen auf die Sequenz und den Aufruf, letzteres ausgedrückt durch das Versenden einer Nachricht an ein Objekt, beschränkt. Alle anderen Kontrollstrukturen müssen mit den Mitteln der Sprache simuliert werden. Was zunächst wie eine erhebliche Einschränkung aussehen mag, erweist sich in der Praxis als gewichtiger Vorteil: Die Programmiererin kann nämlich selbst, wenn ihr danach ist, neue Kontrollstrukturen einführen.

**SMALLTALKS zwei
Kontrollstrukturen**



4.6.1 Verzweigung

Die einfache Verzweigung wird in SMALLTALK durch das Versenden einer Nachricht, die die bedingt auszuführenden Anweisungen in Form eines Blocks als Parameter enthält, an einen Wahrheitswert realisiert:

```
195 x > 5 ifTrue: [Transcript show: "x > 5"]
```

beispielsweise gibt genau dann die Meldung „x > 5“ auf der Konsole aus, wenn `x > 5` zu `true` auswertet.

Um zu verstehen, wie das funktioniert, sehen wir uns zunächst die Implementierung der Verzweigung mittels `ifTrue:` an. Sie wird durch eine Methode `ifTrue: aBlock` realisiert, die als Parameter `aBlock`, also eine Folge von Anweisungen, erhält, das sie, der Bedeutung von `If` entsprechend, entweder ausführt oder eben nicht ausführt. Diese Methode ist für die beiden Objekte `true` und `false` (genauer: für die beiden Objekte, die die Pseudovariablen `true` und `false` benennen; siehe Abschnitt 1.7) jeweils unterschiedlich implementiert:

```
196 ifTrue: aBlock  
197   ^ aBlock value
```

für `true` und

```
198 ifTrue: aBlock  
199   ^ nil
```

für `false`. Wenn also der Empfänger der Nachricht `ifTrue: aBlock` das Objekt `true` ist, dann bewirkt die Auswertung, dass der Block `aBlock` ausgeführt wird; ist das Objekt hingegen `false`, wird `aBlock` nicht ausgeführt, was genau der Bedeutung der Nachricht entspricht. So hat

```
200 1 = 1 ifTrue: [Transcript show: 'Eins bleibt Eins!!!'; cr]
```

genau den erwarteten Effekt. Aufgrund der *Continuation* bei Blöcken (s. Abschnitt 4.4.2) ist es möglich, aus einer Methode mittels `ifTrue:` bedingt zurückzukehren:

```
201 returnEarly  
202 1 = 1 ifTrue: [^ true].  
203   ^ false
```

liefert `true` zurück, da durch die Auswertung des Blocks `[^ true]` per `value` in Zeile 197 nicht der Block, sondern die Methode `returnEarly`, beendet wird. Wie man sieht, hat das zunächst etwas eigenwillig anmutende Konzept der Continuation in Kombination mit Return eine erhebliche programmierpraktische Relevanz.

Aus Symmetriegründen wird auch die Methode `ifFalse: aBlock` angeboten, die wie folgt implementiert ist:



```
204 ifFalse: aBlock  
205   ^ nil
```

für **true** sowie

```
206 ifFalse: aBlock  
207   ^ aBlock value
```

für **false**. Eigentlich viel zu banal, um es hinzuschreiben, aber da es nur ein einziges Mal gemacht werden muss und dabei so etwas Fundamentales wie die Verzweigung realisiert, kann man sich auch daran freuen.

Falls Sie dies dennoch für einen Budenzauber halten, dann haben Sie zumindest nicht völlig unrecht: Die Verzweigung ist nämlich gar nicht wirklich aus dem Sprachkern verschwunden, sie ist nur an einer Stelle versteckt, an der Sie sie vielleicht nicht vermuten: an der Stelle der Auswahl der Methode, die in Reaktion auf den Empfang einer Nachricht ausgeführt wird (also beim *dynamischen Binden*).

**Fallunterscheidung
durch dynamisches
Binden**

Selbsttestaufgabe 4.3

Überlegen Sie, wie Sie das aus anderen Sprachen bekannte If-then-else-Konstrukt in SMALLTALK realisieren würden und schreiben Sie die entsprechenden Methodendefinitionen auf.

Nach demselben Prinzip wie die einfache Fallunterscheidung vermeidet SMALLTALK übrigens auch solche, die das Auftreten von **nil** betreffen. So ist zum Beispiel die Methode **isNil** für **nil** als

```
208 isNil  
209   ^ true
```

und für alle anderen Objekte als

```
210 isNil  
211   ^ false
```

implementiert.

Selbsttestaufgabe 4.4

Überlegen Sie, wie Sie die logischen Operatoren **and:**, **or:** und **not** für **true** und **false** implementieren würden!



4.6.2 Wiederholung

Etwas weiter ausholen müssen wir für die Implementierung von Wiederholungen (Schleifen): Da das Abbruchkriterium von Schleifen immer wieder (bei jedem Schleifendurchlauf) ausgewertet werden muss, kann nicht einfach einmal eine Nachricht an (eine Variable mit Inhalt) `true` oder `false` gesendet werden. Vielmehr muss die Auswertung des Abbruchkriteriums selbst in einem Block stattfinden, der bei jedem Schleifendurchlauf neuerlich ausgewertet wird. Aber auch das ist kein Problem: Der Nachrichtenempfänger ist einfach ein Block, dessen Auswertung entweder `true` oder `false` zurückliefert; der Parameter der Nachricht ist dann der Block, der den Schleifenrumpf darstellt, wie in:

```
212 [ x < 5 ] whileTrue: [ x := x + 1 ]
```

`whileTrue:` ist dazu als Methode für Blöcke wie folgt implementiert (beachten Sie, dass Sie den Empfänger, einen Block, in der Methodendefinition nicht direkt sehen; er wird durch `self` repräsentiert und ist nicht mit dem Parameter `aBlock`, ebenfalls ein Block, zu verwechseln):

```
213 whileTrue: aBlock
214   self value
215   ifTrue: [
216     aBlock value.
217     self whileTrue: aBlock].
218   ^ nil
```

Wie man sieht, wird hier die Schleife durch eine sog. *Endrekursion* simuliert: `whileTrue:` ruft `whileTrue:` am Ende selbst wieder auf. Wegen der Performanz (oder möglicher Beschränkungen der Anzahl der Schleifendurchläufe durch die Größe des Aufrufstacks) braucht man sich dabei keine Sorgen zu machen: Da hinter dem rekursiven Aufruf nichts mehr passiert (deswegen ja Endrekursion), kann dieser vom Compiler in eine echte Schleife übersetzt werden. Für die Implementierung von `whileFalse:` (mit entsprechender Semantik) braucht übrigens nur das `ifTrue:` aus Zeile 215 durch `ifFalse:` ersetzt und der rekursive Aufruf entsprechend angepasst zu werden.

Simulation der Schleife durch Endrekursion

Für die ebenfalls aus anderen Sprachen bekannten For-Schleifen hat SMALLTALK eine andere elegante Lösung parat, auf die wir im nächsten Abschnitt eingehen werden.

4.6.3 Iteration

Wenn Sie If und While schon kennen, kennen Sie sicher auch For. Die klassische Form der For-Schleife verwendet eine Zählvariable, einen Anfangswert, ein Inkrement (das auch negativ, also ein Dekrement sein kann) sowie einen Endwert. Solche For-Schleifen gibt es in SMALLTALK auch:

```
219 5 to: 1 by: -2 do: [ :i | Transcript show: i printString]
```



beispielsweise gibt auf dem Transcript die Folge „531“ aus.

Wir schauen uns den Ausdruck aus Zeile 219 einmal genauer an. Dem

Zählschleife

Objekt 5 wird offenbar eine Nachricht `to:by:do:` gesendet, wobei 5 der Startwert, der Parameter zu `to:`, 1, der Endwert, der zu `by:`, -2, das Inkrement und der zu `do:` ein Block ist. Der Block stellt offenbar, ähnlich wie bei der Realisierung der While-Schleife in SMALLTALK, den Schleifenrumpf dar; er hat einen Parameter `i`, der anscheinend als Zählvariable fungiert. Tatsächlich wird die Methode `to:by:do:` in SMALLTALK EXPRESS wie folgt implementiert:

```
220 to: stop by: step do: aBlock
221   | nextValue |
222   nextValue := self.
223   step = 0 ifTrue: [self error: 'step must be non-zero'].
224   step < 0
225     ifTrue: [[stop <= nextValue]
226       whileTrue:
227         [aBlock value: nextValue.
228          nextValue := nextValue + step]]
229   ifFalse: [[stop >= nextValue]
230     whileTrue:
231       [aBlock value: nextValue.
232          nextValue := nextValue + step]]
```

Hier interessiert uns aber vor allem eine Form der Iteration, die nicht einer einfachen Zählschleife entspricht, sondern über eine Menge von beliebigen Objekten geht. Solche Mengen sind uns ja schon begegnet, wenn auch nur in Gestalt von literalen Arrays.

Anders als in vielen anderen Sprachen kann man in SMALLTALK über die

For-each-Schleife

Elemente eines Arrays direkt, also ohne die Verwendung einer Zählschleife, deren Laufvariable als Index in das Array dient, iterieren. So hat die Auswertung des Ausdrucks

```
233 #(5 3 1) do: [ :i | Transcript show: i printString]
```

exakt das gleiche Ergebnis wie die des Ausdrucks aus Zeile 219, nämlich die Ausgabe von „531“ auf dem Transcript. `i` ist aber diesmal keine Zählvariable, da hier nichts gezählt wird; es ist vielmehr eine Laufvariable, der der Reihe nach die Elemente des literalen Arrays `#(5 3 1)` zugewiesen werden. Dies müssen keine Zahlen sein:

```
234 #'Smalltalk' 'ist' 1.0 'klasse'
235   do: [ :i | Transcript show: i printString]
```

funktioniert genauso. `do:` ersetzt also ganz offensichtlich das aus manchen anderen Sprachen (seit der Version 5.0 auch aus JAVA) bekannte For-each-Konstrukt. Wie wir gleich sehen werden, ist die Iteration, also das Fortschalten der Elemente und die Überprüfung der Abbruchbedingung, in der Collection, über die iteriert wird, implementiert, weswegen man das Verfahren auch **interne Iteration** nennt (in Abgrenzung von der herkömmlichen, **externen Iteration**, bei der die Laufvariable selbst gesetzt und abgefragt werden muss).

interne Iteration



Die Implementierung der Kontrollstruktur erfolgt wiederum selbst in SMALLTALK und ist ziemlich einfach:

Implementierung der
internen Iteration am
Beispiel von do :

```
236 do: aBlock
237   1 to: self size do:
238     [:index | aBlock value: (self at: index)]
```

Dabei ist **to:do:** für Ganzahlen analog zu obigem **to:by:do** implementiert. Die Zählvariable **index** des Blocks von Zeile 238 läuft so von 1 bis zur Anzahl der indizierten Instanzvariablen des Empfängers von **do:** (im obigen Beispiel ein Array), die über den Aufruf von **size** auf dem Empfänger (repräsentiert durch **self**) abgefragt wird. Der Inhalt der indizierten Instanzvariablen des Empfängers wird dann der Reihe nach als Parameter mittels **value:** an den Block **aBlock** zur Auswertung geschickt.

4.6.4 Iterieren über :n-Beziehungen

Bei :1-Beziehungen schickt man häufig dem von der betreffenden Variable referenzierten Objekt eine Nachricht. Der Ausdruck

```
239 freund einladen
```

beispielsweise besagt, dass das Objekt, mit dem der Besitzer der Variable **freund** über diese Variable in :1-Beziehung steht, die Nachricht **einladen** erhalten soll. Wenn man dasselbe mit :n-Beziehungen machen möchte, so erreicht die Nachricht — bei gleicher Vorgehensweise — nicht die logisch in Beziehung stehenden Objekte, sondern das die Beziehung selbst repräsentierende *Zwischenobjekt* (das ja Wert der Variable ist), das mit dieser Nachricht jedoch nichts anfangen kann. Um die Nachricht stattdessen an alle durch das Zwischenobjekt referenzierten Objekte zu senden, schickt man dem Zwischenobjekt die Nachricht **do: aBlock**, wobei **aBlock** ein mit einem Parameter parametrisierter Block ist, der für jedes Element des Arrays genau einmal (mit dem Element als tatsächlichem Parameter) aufgerufen wird. Wenn also z. B. die Instanzvariable **freunde** heißt und eine :n-Beziehung ausdrückt, dann schreibt man statt Zeile 239

```
240 freunde do: [ :freund | freund einladen]
```

Man kann sich fragen, warum die SMALLTALK-Syntax nicht erlaubt, die Nachricht doch direkt an das Zwischenobjekt zu schicken, das die :n-Beziehung repräsentiert (also im gegebenen Beispiel **freunde einladen**), und dies dann intern so umsetzt, dass die Nachricht an alle Objekte geschickt wird. Der einfache Grund dafür wird gleich offenbar: Weil man häufig die Nachricht gar nicht an alle Objekte schicken will, sondern nur an ausgewählte, und weil dazu dann noch weitere Angaben notwendig sind, so dass im allgemeinen Fall nichts gewonnen wäre.

Auf Basis von **do:** lassen sich nun zahlreiche weitere natürliche und äußerst praktische Kontrollstrukturen erzeugen. So ist wie im obigen Beispiel recht häufig eine Nachricht gar nicht an alle Elemente einer :n-Beziehung zu senden,

Iterationen mit
zusätzlicher Funktion

sondern nur an solche, die bestimmte Kriterien erfüllen. Dazu ist es möglich, die Beziehung quasi im Vorübergehen einzuschränken und den Block dann nur auf der Einschränkung auszuführen:

```
241 (freunde select: [ :freund | freund eng == true])
      do: [ :freund | freund einladen]
```

Dabei ist die Methode `select:` wie folgt implementiert:

```
242 select: aBlock
243   | answer |
244   answer := self species new.
245   self do: [ :element |
246     (aBlock value: element)
247       ifTrue: [answer add: element]].
248   ^ answer
```

Zeile 244 müssen Sie hier noch nicht verstehen; der Rest sollte Ihnen aber inzwischen klar sein. `answer` ist eine *temporäre Variable*, die nur innerhalb der Methode Gültigkeit hat; ihr werden in der Do-Schleife mittels `add:` (einer Methode mit offensichtlicher Funktion) alle die Elemente des Empfängers hinzugefügt, für die der Parameterblock `aBlock` zu `true` auswertet.

Auf ähnlich einfache Weise lassen sich nahezu beliebig weitere Kontrollstrukturen realisieren. Zu `select:` komplementär ist beispielsweise die Methode `reject:`, die aus einer `:n-` Beziehung alle die Elemente entfernt, die eine genannte Bedingung nicht erfüllen:

```
249 reject: aBlock
250   ^ self select: [ :element |
251     (aBlock value: element) not]
```

Kaum zu glauben, dass mit so wenig Aufwand der Verwendung eine neue Kontrollstruktur hinzugefügt werden kann.

Eine weitere praktische Methode, die eine Sammlung von Objekten zurückgibt, über die dann (mittels `do:`) iteriert werden kann, ist `collect:`; sie sammelt all die Elemente, die die Auswertung des ihr als Parameter übergebenen Blocks auf den Elementen der ursprünglichen Sammlung zurückliefert, und ist wie folgt implementiert:

```
252 collect: aBlock
253   | answer |
254   answer := self species new.
255   self do: [ :element |
256     answer add: (aBlock value: element)].
257   ^ answer
```

Aber auch einzelne Elemente einer Beziehung lassen sich bestimmen: `detect:` mit einem Block `aBlock` als Parameter aufgerufen liefert z. B. aus einer Sammlung von Elementen das erste Element zurück, auf dem `aBlock` ausgewertet den Wahrheitswert `true` ergibt (wobei bei Fehlen eines solchen Elements ein Fehler geliefert wird). Das erlaubt z. B. den Ausdruck



zu formulieren, der besagt, dass man sich mit dem ersten engen Freund, und nur mit dem, trifft. Für die Praxis wichtiger ist die Variante `detect: aBlock ifNone: exceptionBlock`, die `detect: aBlock` um einen (parameterlosen) Block ergänzt, dessen Wert bei Fehlen eines geeigneten Elements zurückgegeben wird.

Selbsttestaufgabe 4.5

Versuchen Sie, eine Implementierung der Methode `detect: aBlock` selbst zu entwerfen.

Eine ganze Klasse von immer wiederkehrenden Anweisungssequenzen zu ersetzen erlaubt schließlich die Methode `inject:into::`: Es ergänzt die Funktionsweise von `do:` darum, das Ergebnis der Auswertung eines Blocks in einem Iterationsschritt als ersten Parameter in die Auswertung des nächsten Schritts einzuspeisen. So lässt sich beispielsweise das öde Akkumulieren von Eigenschaften (inkl. der gern vergessenen Initialisierung des Akkumulators) elegant wie folgt ausdrücken:

Kumulation

```
259 freunde
260   inject: true
261   into: [ :einsam :freund | einsam and: [freund eng not]]
```

So einfach kann Programmieren sein!

Für Interessierte: Methoden wie `do:`, `collect:`, `select:` und `inject:` sind allesamt (als Funktionen höherer Ordnung) aus der funktionalen Programmierung bekannt. In die objektorientierte Programmierung mit Sprachen wie JAVA oder C# haben sie jedoch erst sehr spät Einzug gehalten.



5 Zusammenfassung der SMALLTALK-Syntax

Sicher ist Ihnen aufgefallen, dass uns bislang keine *Schlüsselwörter* in SMALLTALK begegnet sind (bis auf die sog. *Schlüsselwortnachrichten*, die aber frei wählbar und deswegen eben gerade keine Schlüsselwörter sind; vgl. Abschnitt 4.1.2). Der Grund hierfür ist einfach: Es gibt keine *Schlüsselwörter*, lediglich ein paar *Symbole mit spezieller Bedeutung*. Es sind dies:

Symbol	Bedeutung/Verwendung
<code>:=</code>	Zuweisung
<code>.</code>	Trennzeichen zwischen zwei Anweisungen sowie Dezimalpunkt für Gleitkommazahlen
<code>;</code>	Trennzeichen zum Kaskadieren von Nachrichten
<code>:</code>	Markierung von Parametern in Nachrichten und Blöcken
<code>()</code>	Klammerung von Ausdrücken zur Festschreibung der Reihenfolge der Auswertung
<code>[]</code>	Bildung von Blöcken
<code> </code>	Trennzeichen zwischen den Parametern eines Blocks und seinen Anweisungen



Symbol	Bedeutung/Verwendung
	Deklaration von temporären Variablen in Methoden (und Blöcken)
'' ''	Markierung von Kommentaren
' '	Markierung von String-Literalen
\$	Markierung von Zeichenliteralen
#	Markierung von Symbol- und Array-Literalen
^	Rückgabe-Operator (Return)

Das ist alles! Die reservierten Namen `true`, `false`, `nil`, `self` und `super` sind die von *Pseudovariablen*; alle aus anderen Sprachen bekannten Schlüsselwörter sind als Methoden in SMALLTALK selbst definiert.

6 Lösungen zu den Selbsttestaufgaben

Selbsttestaufgabe 1.1 (Seite 17)

```
262 $a == $a ⇒ true
263 1 == 1 ⇒ true
264 1.0 == 1.0 ⇒ false (true in SQUEAK)
265 100000000 == 100000000 ⇒ false (true in SQUEAK)
266 #(1 2 3) == #(1 2 3) ⇒ false (true in SQUEAK)
```

Es fällt auf, dass bei manchen Zahlen (syntaktisch) gleiche Literale identische Objekte repräsentieren, manche nicht. In SQUEAK ist das übrigens nur anders, weil der Compiler eine Art „*Literaloptimierung*“ betreibt: Zwei gleiche Literale, die gemeinsam übersetzt werden, liefern das identische Objekt.

Selbsttestaufgabe 1.3 (Seite 23)

Das ist im wesentlichen dieselbe Aufgabe wie Selbsttestaufgabe 1.1, also auch dieselbe Lösung.

Dies erklärt auf einfache Weise, warum der Test auf Identität (`==`) bei gleichen Literalen für verschiedene Arten von Objekten zu verschiedenen Ergebnissen führt: Er vergleicht einfach die systeminterne Repräsentation. Handelt es sich bei der linken und der rechten Seite um den Verweis auf dieselbe Stelle im Speicher, ergibt der Test `true`; handelt es sich bei der linken und der rechten Seite um denselben Wert (im Falle von Zeichen und kleinen Ganzzahlen), ergibt der Test ebenfalls `true`. In allen anderen Fällen ergibt er `false`.

Selbsttestaufgabe 1.2 (Seite 21)

Wie SQUEAK so nett sagt: „lots of globals“, so u. a. `Transcript` und sich selbst. Zugleich enthält es offenbar auch ein Symbol für jede Klasse (so z. B. `#Object`).



Selbsttestaufgabe 4.1 (Seite 50)

implizit: g := e, f := 2

explizit: a := f

Selbsttestaufgabe 4.2 (Seite 62)

(a) Da die bei der Auswertung des Blocks implizite Zuweisung x := a bzw. y := b Zeiger auf Objekte und nicht, wie beim *Call by reference*, Zeiger auf Variablen überträgt, ändert eine Änderung des Inhalts von x und y nicht auch den Wert von a und b. Siehe auch die Bemerkungen zu *Call by reference vs. Call by value* in Abschnitt 4.3.2.

(b) Das Problem ergibt sich beim parameterlose Block nicht, da hier direkt die Variablen aus dem *Home context* verwendet werden.

(c) Weil man naiverweise davon ausgehen könnte, dass die Zuweisung an formale Parametervariablen eben auch die tatsächlichen Parametervariablen betrifft. Es ist daher sinnvoll, die Zuweisung an formale Parameter grundsätzlich zu verbieten, um der Programmiererin die Enttäuschung zu ersparen, wenn sie feststellen muss, dass es auf die tatsächlichen Parametervariablen keinen Effekt hat.

Selbsttestaufgabe 4.3 (Seite 65)

Einfach einen Block-Parameter anhängen:

für true:

```
267 ifTrue: wahrBlock else: falschBlock  
268     ^ wahrBlock value
```

für false:

```
269 ifTrue: wahrBlock else: falschBlock  
270     ^ falschBlock value
```

Selbsttestaufgabe 4.4 (Seite 65)

für true:

```
271 or: arg  
272     ^ true  
  
273 and: arg  
274     ^ arg  
  
275 not
```



276 ^ false

für false: analog

Selbsttestaufgabe 4.5 (Seite 70)

277 **detect: einBlock**

278 self do: [:elem | (einBlock value: elem) ifTrue: [^ elem]]



Kurseinheit 2: Systematik der objektorientierten Programmierung

In der vorangegangenen Kurseinheit haben Sie die Grundkonzepte der objektorientierten Programmierung mit SMALLTALK kennengelernt. Neben den Objekten selbst zählen dazu vor allem die Beziehungen zwischen diesen (durch Instanzvariablen ausgedrückt), der davon abgeleitete Zustand von Objekten sowie das in Form von Methoden definierte Verhalten. Was bislang verschwiegen wurde, ist, wie Objekte, für die es keine literale Repräsentation gibt, entstehen und wie ihnen ihre Instanzvariablen und ihre Methoden zugeordnet werden.

Wie das zu geschehen hat, ist mit dem Begriff der objektorientierten Programmierung nicht grundsätzlich festgelegt. Eine gewisse Anerkennung und Verbreitung erfahren haben aber drei verschiedene Ansätze:

Alternativen der Objekterzeugung

1. der Konstruktoransatz, bei dem der Aufbau eines Objekts in einer Methode beschrieben wird, in der dem Objekt bei seiner Erzeugung Instanzvariablen und Methoden zugeordnet werden; verschiedene Konstruktoren erzeugen dann verschiedene aufgebaute Objekte;
2. der Prototypenansatz, bei dem ein schon existierendes Objekt samt seiner Instanzvariablen und Methoden geklont wird; ein Klon kann bei Bedarf um weitere Instanzvariablen und Methoden ergänzt werden oder geklonte können abgeändert oder entfernt werden; und
3. der Klassenansatz, bei dem alle Objekte als *Instanzen* von bestimmten Vorlagen, die entweder selbst keine Objekte oder Objekte auf einer anderen Ebene sind, erzeugt werden.

Den Konstruktoransatz findet man in Sprachen wie EMERALD, den Prototypenansatz in Sprachen wie SELF oder JAVASCRIPT und den Klassenansatz in Sprachen wie SMALLTALK, C++, EIFEL, JAVA, C#, SCALA und vielen anderen mehr.



Aus verschiedenen Gründen hat sich die dritte Variante, die **klassenbasierte Form der Objektorientierung** (wobei die Klassen die erwähnten Vorlagen sind) gegenüber der zweiten, der **prototypenbasierten Form der Objektorientierung** weitgehend durchgesetzt. Die erste Variante findet im Zuge einer gewissen Ernüchterung bzgl. der objektorientierten Programmierung zunehmend Anhänger, und zwar da, wo Objekte und *dynamisches Binden* (s. Abschnitt 12) im Kontext traditioneller imperativer Programmierung angeboten werden sollen. Sie liegt damit aber außerhalb des Gegenstands dieses Kurstextes.



Die Dominanz der klassenbasierten Form der objektorientierten Programmierung liegt vermutlich zum einen daran, dass Klassen ein klassisches, in anderen Disziplinen wie der Mathematik oder der Biologie fest etabliertes Ordnungskonzept darstellen, mit dessen Hilfe sich auch objektorientierte Programme gut strukturieren lassen, und zum anderen daran, dass Klassen sich als (Vorlagen für) Typen eignen und somit die objektorientierte Programmierung Eigenschaften anderer, nicht objektorientierter, dafür aber typisierter Sprachen (also Sprachen, bei denen alle Variablen und Funktionen bei der Deklaration einen Typ zugeordnet bekommen und der Variableninhalt immer vom deklarierten Typ sein muss) übernehmen kann (der Gegenstand von Kurseinheit 3). Die prototypenbasierte Form der Objektorientierung hat hingegen den Charme, dass sie mit weniger Konzepten auskommt und dass sie sehr viel flexibler einzelne Objekte an ihren jeweiligen Zweck anpassen kann, z. B. indem sie eine Methodendefinition nur für ein einziges Objekt abzuändern erlaubt. Letzteres ist z. B. bei der Programmierung von grafischen Bedienoberflächen, bei der das Drücken verschiedener Buttons jeweils verschiedene Ereignisse auslöst (Methoden aufruft), sehr praktisch. Nicht umsonst ist JAVASCRIPT als Programmiersprache für interaktive Webseiten so erfolgreich.

Auch wenn es gute Gründe für die prototypenbasierte Form der objektorientierten Programmierung gibt (und sich deswegen jedes Werk zum Thema objektorientierte Programmierung — so wie auch das Ihnen vorliegende — gemüßigt fühlt, darauf hinzuweisen, dass es sie gibt), werden ich mich im folgenden vornehmlich Klassen als strukturbildenden Konzepten der objektorientierten Programmierung zuwenden und nur hier und da Prototypen kurz die Referenz erweisen.²⁸

7 Klassen

Sprachphilosophisch gesehen ist eine Klasse ein *Allgemeinbegriff* wie etwa *Person*, *Haus* oder *Dokument*. Diese Allgemeinbegriffe stehen in der Regel für eine ganze Menge von Objekten, also etwa alle Personen, Häuser oder Dokumente. Gleichwohl ist die Klasse selbst immer ein Singular — sie ist nämlich selbst ein Objekt, das unter den Allgemeinbegriff *Klasse* fällt. Diese Sprachregelung wird auch in der objektorientierten Programmierung eingehalten (obwohl sie natürlich nicht, da Computer unsere Sprache nicht kennen, überprüft werden kann und deswegen Abweichungen immer wieder vorkommen): Alle Klassennamen sind Singulare.

²⁸ Interessanterweise war der prototypenbasierten Sprache SELF bei SUN MICROSYSTEMS ursprünglich der Platz zugedacht, den dann später JAVA einnehmen sollte. SELF war zwar für die Entwicklung der Java Virtual Machine ein wichtiger Ideenlieferant, ist jedoch außerhalb dieser Kreise kein Erfolg beschieden gewesen. Der Erfolg von JAVASCRIPT zeigt aber, dass das Konzept der prototypenbasierten Programmierung zumindest kein Irrweg war.





Mit jedem Allgemeinbegriff verbinden wir eine ganze Reihe von *Eigenschaften*, die für ihn charakteristisch sind, die wir aber nicht dem Begriff selbst, sondern den Objekten, die darunter fallen, zuordnen. Mit *Person* etwa ist *Name* verbunden sowie *Geburtstag* und ggf. weitere Attribute, aber auch bestimmtes, für Personen charakteristisches *Verhalten*. Das gleiche gilt für *Haus*, *Dokument* und alle anderen Allgemeinbegriffe. Existenz und Adäquatheit von Allgemeinbegriffen sind Thema großer philosophischer Diskurse wie etwa dem sog. *Universalienstreit* und stehen hier nicht zur Debatte. Wichtig ist, dass mit ihnen stets Sätze wie „<ein Individuum> ist ein <ein Allgemeinbegriff>“ gebildet werden können, also etwa „Peter ist eine Person“. Mit solchen Sätzen verbindet sich nämlich die Übertragung aller Eigenschaften, die mit einem Allgemeinbegriff verbunden sind (s. o.), auf das Individuum. So hat Peter, wenn er eine Person ist und *Person* wie oben definiert wurde, eben auch einen Namen und einen Geburtstag.

7.1 Klassifikation

Durch die Zuordnung von Individuen oder Objekten zu Allgemeinbegriffen oder Klassen findet eine **Klassifikation** statt. Diese Klassifikation erlaubt eine Ordnung oder Strukturierung der Anwendungsdomäne, indem bestimmte Aussagen nur noch für die Klassen getroffen werden müssen und nicht mehr für jedes einzelne Objekt. Anstatt also wie in der vorangegangenen Kurseinheit Instanzvariablen und Methoden direkt Objekten zuzuordnen, verbindet man sie mit Klassen und vereinbart, dass alle mit einer Klasse verbundenen Eigenschaften und Verhaltensspezifikationen nicht die Klasse in ihrer Gesamtheit, sondern die einzelnen Objekte, die zu der Klasse gehören, beschreiben.

In diesem Zusammenhang ist es sinnvoll, die Unterscheidung von Extension und Intension eines Begriffs ins Spiel zu bringen. Unter der **Extension** (Ausdehnung oder Erstreckung) eines (Allgemein-)Begriffs versteht man die Menge der Objekte, die darunterfallen. Im Fall von *Person* etwa ist das die Menge aller Personen, im Fall von *Dokument* die Menge aller Dokumente. Die **Intension** (nicht zu verwechseln mit Intention!) eines (Allgemein-)Begriffs hingegen ist die Summe der Merkmale, die den Begriff ausmachen und die die Objekte, die darunter fallen, charakterisieren. Sie ist gewissermaßen das Auswahlprädikat oder die charakteristische Funktion, die zu einem beliebigen Element entscheidet, ob es unter den Begriff fällt. Schon Aristoteles fiel auf, dass Intension und Extension in einem inversen Größenverhältnis zueinanderstehen: Mit wachsender Intension schrumpft die Extension und umgekehrt. Dies ist freilich nicht weiter verwunderlich: Je umfangreicher die Charakterisierung einer Menge von Objekten ist, d. h., je strenger die Bedingungen sind, die ein Objekt erfüllen muss, um dazugehören, desto weniger Objekte erfüllen diese Bedingungen und desto kleiner ist entsprechend die Menge. Wir werden in Kapitel 9 noch einmal darauf zurückkommen.



Allgemeinbegriffe sind die Vorbilder für Klassen in der objektorientierten Programmierung. Interessanterweise bildet eine wichtige philosophische

Allgemeinbegriff vs.
Familienähnlichkeit

Abweichung vom Glauben an die Adäquatheit von Allgemeinbegriffen, die Idee der *Familienähnlichkeiten*, wie sie von Ludwig Wittgenstein in seiner späten Philosophie entworfen wurde, die Grundlage für die schon erwähnte Alternative zu den Klassen, nämlich die **Prototypen**.



WIKIPEDIA

Die Idee Wittgensteins, wie auch der prototypenbasierten Programmierung, ist, dass ein Allgemeinbegriff (eine Klasse) niemals eine adäquate Beschreibung aller Individuen (Objekte) sein kann, die man mit dem Begriff verbindet. Wittgenstein verwendet dafür das Beispiel vom Spiel: Auch wenn es Spiele gibt, die einander stark ähneln, so ist der Begriff vom Spiel doch nicht so scharf gefasst, dass es eine Grenze gäbe, innerhalb derer etwas ein Spiel wie alle anderen, außerhalb derer es aber kein Spiel mehr ist. Vielmehr gibt es, nach Wittgenstein, mehr oder weniger „spielhafte“ Spiele, also prototypische Spiele und solche, die diesen mehr oder weniger gleichen.

Zwar gibt es Anwendungsdomänen, in denen Wittgensteins Familienähnlichkeiten die Sachlage besser beschreiben als die traditionellen Allgemeinbegriffe (man denke z. B. an Musik, in der es zwar Töne und Noten gibt, aber dennoch

Zweckmäßigkeit des
Klassenbegriffs

zwei Töne selten genau gleich klingen sollen und die Notenzeichen entsprechend vielfältig variieren), aber insgesamt sind die üblichen Anwendungen doch eher der Natur, dass es von einigen wenigen Sorten eine große Menge von Objekten gibt, die alle mehr oder weniger gleich zu behandeln sind. Und so vereinfachen Allgemeinbegriffe, oder Klassen, unsere Weltsicht ganz erheblich und damit auch die Programme, die wir schreiben, um unsere Weltsicht zu reflektieren.

Nachdem wir uns also auf Klassen festgelegt haben, können wir nun endlich zur Lüftung des Geheimnisses kommen, wo in einem SMALLTALK-Programm die Instanzvariablen und Methoden, die Objekte ihr eigen nennen, vereinbart (deklariert) und im Falle der Methoden auch definiert (mit Inhalt versehen) werden: in Klassen.

7.2 Klassendefinitionen

Eine **Klassendefinition** liefert die *Intension* einer Klasse. Sie besteht in SMALLTALK zunächst aus der Angabe eines nicht anderweitig verwendeten, durch ein Symbol repräsentierten Klassennamens sowie der Angabe der die Objekte der Klasse beschreibenden *Instanzvariablen* und *Methodendefinitionen*. Anders als in vielen anderen objektorientierten Programmiersprachen erfolgt in SMALLTALK die Klassendefinition nicht in einer Datei (was hätte eine Datei auch mit den Konzepten einer Programmiersprache zu tun?), sondern durch Eintragungen in eine dafür vorgesehene Datenstruktur (genauer: durch Erzeugung eines die Klasse beschreibenden Objekts). Es gibt also auch insbesondere keine Syntax für eine Klassendefinition, sondern nur ein Schema. Ein solches, wenn auch noch unvollständiges, Schema ist das folgende:

Schema einer
Klassendefinition



Klasse	<Klassenname>
benannte Instanzvariablen	<Liste von Instanzvariablennamen>
indizierte Instanzvariablen	<ja/nein>
atomar	<ja/nein>
Instanzmethoden	<Liste von Methodendefinitionen>

Alle Instanzen einer Klasse verfügen somit über den gleichen Satz von Instanzvariablen, aber nicht denselben, was soviel bedeutet wie dass jede Instanz der Klasse (jedes Objekt, das zur Extension der Klasse gehört) diese Variablen individuell belegen kann. Im Gegensatz dazu verstehen alle Instanzen einer Klasse nicht nur dieselben Nachrichten, sie verwenden auch dieselben Methodendefinitionen, um auf die Nachrichten zu reagieren. Instanzen einer Klasse können sich also nur insoweit in ihrem Verhalten unterscheiden, wie sich die Methodendefinitionen auf die Werte der Instanzvariablen beziehen, wie also das in den Methoden spezifizierte Verhalten vom Inhalt der Instanzvariablen abhängt. Insbesondere ist es nicht vorgesehen, dass verschiedene Instanzen *einer* Klasse über verschiedene Definitionen *einer* Methode (genauer: über verschiedene Definitionen von zu der Nachricht passenden Methoden) verfügen. Das unterscheidet die klassenbasierte von der prototypenbasierten Form der objektorientierten Programmierung.

Die beiden Einträge „indizierte Instanzvariablen“ und „atomar“ stehen übrigens dafür, ob eine Instanz der Klasse indizierte Instanzvariablen haben soll (klar) und falls ja, ob diese Variablen dann eine binäre Repräsentation (ja) oder Referenzen (nein) enthalten. Mit Hilfe von indizierten Instanzvariablen, die binäre Repräsentationen enthalten, werden z. B. Zahlen, Strings, aber auch Bitmaps wie Fensterinhalte, der Cursor oder Fonts intern dargestellt. Da man als Programmiererin solche Klassen in der Regel nicht selbst anlegt, werde ich den Eintrag „atomar“ zukünftig unter den Tisch fallen lassen.

Eine Klasse (das Objekt, das die Klassen repräsentiert, nicht ihr Name) wird in SMALLTALK nach ihrer Erzeugung übrigens durch eine *globale Pseudovariable* repräsentiert, deren Name der Name der Klasse ist. Da die Variable global ist,

Repräsentation von Klassen im System

muss sie (und damit auch der Name der Klasse) immer mit einem Großbuchstaben beginnen (s. Abschnitt 1.5.2 in Kurseinheit 1). Die Variable wird automatisch mit der Klassendefinition eingeführt (vereinbart); ihr „Wert“, die Klasse, auf die sie verweist, ist das Objekt, das ihr bei der Anlage der Klasse zugewiesen wird. Da Klassennamen globale Variablen sind, da sie insbesondere absolut global sind und nicht nur in Bezug auf irgendeine Programmeinheit (wie etwa eine Methodendefinition), sind sie von überall her zugreifbar. Außerdem wird jede neue Klasse in eine Art Systemwörterbuch namens „Smalltalk“ (repräsentiert von der globalen Variable **Smalltalk**; s. Selbsttestaufgabe 1.2) eingetragen und ihr Name (als Symbol) in die Symboltabelle **SymbolTable**.



Selbsttestaufgabe 7.1

Vergewissern Sie sich, dass die Klasse **Class** in **Smalltalk** enthalten ist und das Symbol **#Class** in **SymbolTable** (nur SMALLTALK EXPRESS). Enthält **Smalltalk** auch andere Objekte als Klassen?

Mittels einer solchen Klassendefinition ist man nun in der Lage, das SMALLTALK-System um neue, eigene Klassen zu erweitern. Ein Beispiel für eine solche neue Klasse gibt die folgende (wie gesagt noch unvollständige) Klassendefinition, die auf einfache Weise einen Stapspeicher (Stack) implementiert, der seine Elemente in indizierten Instanzvariablen und den Stapelzeiger (Stack pointer) in einer benannten hält:

**Erweiterung von
SMALLTALK um eine
Beispielklasse**

Klasse	Stack
benannte Instanzvariablen	stackpointer
indizierte Instanzvariablen	ja
Instanzmethoden	

```
279 push: anElement
280     "legt neues Element auf Stapel"
281     stackpointer := stackpointer + 1.
282     self at: stackpointer put: anElement

283 pop
284     "entfernt oberstes Element vom Stapel"
285     stackpointer := stackpointer - 1

286 top
287     "liefert oberstes Element des Staps"
288     ^ self at: stackpointer
```

Nur zur Wiederholung: Die Pseudovariable **self** in den Zeilen 282 und 288 steht jeweils für das Objekt, das die die Methode auslösende Nachricht erhalten hat (das Empfängerobjekt): Sie ist notwendig, da der Zugriff auf die indizierten Instanzvariablen in SMALLTALK immer über die Methoden **at:** und **at:put:** erfolgt, deren Aufruf (als Nachrichtenausdruck) stets einen Empfänger benötigt. Anders als z. B. in JAVA (wo **this** die Rolle von **self** einnimmt) wird bei fehlendem Empfänger innerhalb einer Methode nicht einfach das Empfängerobjekt angenommen, sondern ein Syntaxfehler gemeldet.

Bei genauem Hinsehen bemerkt man, dass die obige Implementierung eines Stacks einen Schönheitsfehler besitzt: Während die Manipulation der benannten Instanzvariable **stackpointer**, deren Wert ja von den Methoden von **Stack** aktualisiert wird, durch andere Objekte noch verhindert werden kann, ist dies für die indizierten Instanzvariablen eines Stack-Objekts nicht der Fall. Eine Benutzerin eines solchen Objekts kann stattdessen mittels **at:** und **at:put:** jederzeit auf jedes beliebige Element des Stacks zugreifen, und zwar unabhängig davon, ob dies gerade

**Repräsentation des
Stack-Inhalts
verbergen**



oben auf dem Stack liegt. Eine Instanz der Klasse **Stack** verbirgt also nicht wie in Kurseinheit 1, Abschnitt 4.3.4 gefordert die Repräsentation ihres Zustands, der Stack-Elemente. Um dies zu bewirken, muss man anstelle der indizierten Instanzvariablen eine benannte verwenden, die selbst ein Objekt mit indizierten Instanzvariablen hält (ein *Zwischenobjekt*; s. Abschnitt 2.2), und die Speicherung der Elemente des Stacks diesem zweiten Objekt übertragen. Die Implementierung sähe dann wie folgt aus:

Klasse	Stack
benannte Instanzvariablen	stackcontent stackpointer
indizierte Instanzvariablen	nein
Instanzmethoden	
289	push: anElement
290	"legt neues Element auf Stapel"
291	stackpointer := stackpointer + 1.
292	stackcontent at: stackpointer put: anElement
293	pop
294	"entfernt oberstes Element vom Stapel"
295	stackpointer := stackpointer - 1
296	top
297	"liefert oberstes Element des Stapels"
298	^ stackcontent at: stackpointer

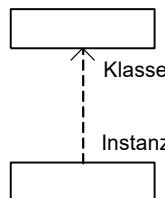
Auf die Variable **stackcontent** kann von anderen Objekten nicht direkt zugegriffen werden — sie ist verborgen (und nur noch indirekt, über **push:**, **pop** und **top** manipulier- bzw. lesbar). Das bedeutet jedoch nicht, dass auf das von **stackcontent** benannte (Zwischen-)Objekt nicht zugegriffen werden kann — aufgrund der oben dargestellten Klassendefinition ist nämlich noch unklar, wo das Objekt, das den Stack-Inhalt fasst, herkommt, so dass nicht ausgeschlossen werden kann, dass bereits *Aliase* existieren (s. Abschnitt 1.8). Eine Möglichkeit, dies auszuschließen, ist, das Stack-Objekt sein Zwischenobjekt selbst erzeugen zu lassen. Dem wenden wir uns als nächstes zu.

7.3 Instanziierung

Klassendefinitionen bilden also eine Art Vorlage für Objekte. Um nun von einer solchen Vorlage Objekte mit Eigenschaften (Instanzvariablen und Methoden), wie sie durch die Definition (Intension) festgelegt sind, zu erzeugen, muss man sie **instanziieren**. Die Instanziierung ist ein Vorgang, bei dem ein neues Objekt entsteht — sie ist gewissermaßen die Umkehrung der *Klassifikation*, also des Übergangs vom Individuum zu seiner Klasse (wobei die Klassifikation, anders als die Instanziierung, in der Programmierung kein Vorgang ist). Vom erzeugten Objekt sagt man dann, es sei **Instanz** dieser Klasse. Tatsächlich spricht man in SMALLTALK, da ja auch Klassen Objekte sind, häufig von Instanzen anstelle von Objekten, wenn keine Klassen gemeint sind. Wie wir schon im nächsten Kapitel sehen werden, sind in SMALLTALK jedoch auch Klassen Instanzen. So gesehen



handelt es sich bei den Begriffen *Instanz* und *Klasse* also eher um Rollen von Objekten, die im Verhältnis der Instanziierung zueinander stehen. Die Begriffsbildung der objektorientierten Programmierung ist an dieser Stelle aber leider nicht besonders gelungen.²⁹



In SMALLTALK ist jedes Objekt Instanz genau einer Klasse. (Genaugenommen ist es **direkte Instanz** genau einer Klasse, aber zum Unterschied zu *indirekten Instanzen* kommen wir erst in Abschnitt 9.1.) Dabei weiß jedes Objekt, von welcher Klasse es eine Instanz ist; diese Information lässt sich dem Objekt durch Senden der Nachricht `class` entlocken; der entsprechende Ausdruck liefert das Objekt, das die Klasse repräsentiert, zurück.

Objekte kennen ihre Klassen

Mit der Instanziierung wird der *Extension* einer Klasse ein neues Element hinzugefügt. Das Elementsein auf Mengenebene entspricht also in etwa dem Instanzsein auf programmiersprachlicher Ebene (in *UML* wie im obigen Diagramm durch einen gestrichelten Pfeil angedeutet). Wir werden noch öfter auf diese mengentheoretische Interpretation zurückkommen.

mengentheoretische Interpretation

Die Objekte, die wir in Kurseinheit 1 kennengelernt haben, wurden sämtlich durch *Literale* repräsentiert; diese Objekte sind, da sie vom Compiler erzeugt werden, aus Sicht des Programms „schon da“, wenn es ausgeführt wird.³⁰ Mittels Instanziierung und Klassen hat man nun die Möglichkeit, neue — und neuartige — Objekte programmatisch, also per Programmausführung, zu erzeugen. Dies geschieht standardmäßig, indem man der Klasse, von der man eine Instanz haben möchte, die Nachricht `new` (für Klassen ohne indizierte Instanzvariablen) oder `new:` (für Klassen mit indizierten Instanzvariablen) schickt. Das neue Objekt wird in Reaktion auf die Nachricht (durch eine entsprechende *primitive Methode* des SMALLTALK-Systems; s. Abschnitt 4.3.7) im Speicher angelegt und seine Instanzvariablen werden alle mit `nil` initialisiert. Der Parameter der Nachricht

Instanziierung mit new und new:

²⁹ Auch wenn in SMALLTALK alles ein Objekt ist, gibt es in SMALLTALK doch zwei verschiedene Arten von Objekten, nämlich solche, die instanzierbar sind, und solche, die es nicht sind. Instanzierbare Objekte sind immer (auch) Klassen; für solche, die es nicht sind, gibt es leider keine spezielle Bezeichnung. Man könnte sie jedoch, wie wir in Abschnitt 8.5 sehen werden, Ebene-0-Objekte nennen.

³⁰ Dabei sind die Klassen, von denen es solche Objekte gibt, insofern privilegiert, als sie der Compiler kennen muss, damit er ihnen die Literale (anhand ihrer Syntax) zuordnen kann, um die richtigen Objekte (Objekte der richtigen Klassen) zu erzeugen. Da Klassen (zumindest in SMALLTALK-80 und allen davon abgeleiteten Dialekten) keine Schnittstelle zum Compiler haben, ist es auch nicht möglich, für selbst definierte Klassen eigene literale Objektrepräsentationen zu kreieren.



new: muss immer eine natürliche Zahl sein und legt die Anzahl der indizierten Instanzvariablen fest, über die ein Objekt verfügt. Hat ein Objekt (per Klassendefinition; s. Abschnitt 7.2) keine indizierten Instanzvariablen, führt **new**: zu einem Laufzeitfehler.

Einen neuen Stack mit Platz für 100 Elemente erhält man, indem man, bei obiger erster Klassendefinition von **Stack**, den Ausdruck

Beispiel

299 **Stack new: 100**

auswertet. Dabei ist **Stack** wie gesagt eine Pseudovariable, die den Klassennamen trägt, die auf das Objekt verweist, das die Klasse repräsentiert, und der ihr Wert beim Anlegen der Klasse vom System zugewiesen wurde. In der zweiten Form der Implementierung wäre eben dieser Ausdruck verboten: Stattdessen dürfte der Ausdruck dann nur noch

300 **Stack new**

heißen. Dass der Stack dann trotzdem 100 Elemente halten kann, muss in diesem Fall bei der Instanziierung des Zwischenobjekts, auf das die Variable **stackcontent** verweist, mittels eines entsprechenden New-Ausdrucks angegeben werden. Diese Instanziierung hatten wir jedoch oben unterschlagen; wo und wie sie durchgeführt wird, wird Gegenstand des nächsten Kapitels sein, wenn es um Konstruktoren und Initialisierung geht.

Eine Alternative zum Instanziieren ist übrigens das **Klonen**. Beim Klonen wird ein neues Objekt auf der Basis eines bereits existierenden erzeugt. Der Klon stellt also eine Kopie dar. Beim Klonvorgang ist festzulegen, wie

**Klonen als
Alternative zum
Instanziieren**

weit (tief) das Kopieren gehen soll, also ob nur das Objekt oder auch seine Attributobjekte und die, zu denen es in Beziehung steht (auf die die Instanzvariablen verweisen; s. Kapitel 2 in Kurseinheit 1) kopiert werden sollen. Während in prototypenbasierten objektorientierten Programmiersprachen, die das Konzept der Klasse ja nicht kennen, Klonen die einzige Möglichkeit ist, neue Objekte zu erstellen, müssen bei Programmiersprachen mit Klassen, in denen jedes Objekt Instanz einer Klasse sein muss, Klone in der Regel durch Instanziierung und Übertragung der Inhalte der Instanzvariablen erzeugt werden. Da wir hier aber die klassenbasierte Linie verfolgen und auf klassenlose objektorientierte Programmiersprachen nur eingehen, wo dies interessant erscheint, werden wir das Klonen, das in klassenbasierten objektorientierten Programmiersprachen eine untergeordnete Rolle spielt, erst in Abschnitt 14.1 vertiefen.

8 Metaklassen

Da in SMALLTALK auch eine Klasse ein Objekt ist, kann die Klasse selbst, genau wie alle anderen Objekte, Instanzvariablen und -methoden haben. Aber wo werden diese definiert? Der Analogie der Objekte, die Instanzen der Klasse sind, folgend müsste das in der jeweiligen Klasse der Klasse, also der Klasse, von der die Klasse (als Objekt) eine Instanz ist, erfolgen. Und so ist es tatsächlich auch.



Zunächst könnte man annehmen, dass alle Klassen Instanzen einer speziellen Klasse, nennen wir sie **Class**, sind. Jede Klasse hätte dann (als Instanz dieser Klasse) die Instanzvariablen und Methoden, die in **Class** definiert sind. Insbesondere hätte jede Klasse dieselbe Menge von Instanzvariablen und Methoden. Dies scheint zunächst auch sinnvoll, denn bei den Klassen handelt es sich ja um Objekte derselben Art, nämlich einheitlich um Klassen.

Es stellt sich dann die Frage, welche die Instanzvariablen und Methoden, die alle Klassen gleichermaßen charakterisieren, sein könnten. Es könnte z. B. jede Klasse eine Instanzvariable haben, die alle von der Klasse instanzierten Objekte enthält, sowie eine weitere, die diese Objekte zählt.³¹ Eine typische Methode jeder Klasse wäre z. B. **new**, die eine neue Instanz dieser Klasse zurückgibt. Was aber, wenn man weitere Eigenschaften (Instanzvariablen oder Methoden) für eine Klasse haben möchte, die diese nicht mit allen anderen teilt? Was, wenn man eine Methode wie z. B. **new** für eine Klasse anders definieren will als für andere? Im Fall von **new** z. B. ist es denkbar, dass man sie für bestimmte Klassen so umschreiben möchte, dass die Instanzvariablen der neu erzeugten Instanzen bestimmte Startwerte zugewiesen bekommen (so wie die eine oder andere es vielleicht von den Konstruktoren von C++, JAVA oder C# schon kennt und wie es beim Beispiel mit **Stack** oben natürlich gewesen wäre).

Tatsächlich hat die Programmierpraxis gezeigt, dass es günstig ist, wenn jede Klasse (als Instanz) ihre eigenen Instanzvariablen und Methoden besitzt und wenn die Programmiererin diese jeweils frei bestimmen kann, ohne dabei gleichzeitig an andere Klassen denken zu müssen. Um dies zu ermöglichen, muss aber jede Klasse Instanz einer eigenen Klasse sein, in der diese Variablen und Methoden nur für sie angelegt werden können. Und genau das ist in SMALLTALK der Fall.³²

Zu jeder Klasse des SMALLTALK-Systems gehört nämlich genau eine Klasse, von der erstere (und nur diese) eine Instanz ist. Diese zweite Klasse wird **Metaklasse** der ersten genannt. Da eine 1:1-Beziehung zwischen Klassen und ihren Metaklassen besteht, ist es nicht sinnvoll, ihre Benennung den Programmierinnen zu überlassen; sie wird in SMALLTALK stets durch den Ausdruck **<Klassename> class**, also beispielsweise **Stack class**, bezeichnet. Daraus folgt bereits, dass die Programmiererin die Metaklasse nicht selbst anlegen muss (denn dabei müsste sie ja auch einen Namen vergeben) — sie wird vielmehr automatisch mit angelegt, wenn die Programmiererin eine neue Klasse definiert.

Im Prinzip ist die Definition einer Metaklasse genauso aufgebaut wie die einer normalen Klasse: Sie besteht aus der Angabe einer Menge von be-

³¹ Tatsächlich gibt es solche Variablen. Sie können ja mal zum Spaß versuchen, sie zu finden.

³² In früheren Versionen SMALLTALKS war das übrigens nicht so und ALAN KAY, der das Projekt bereits vor der Veröffentlichung von SMALLTALK-80 verlassen hatte, ist selbst einer der größten Kritiker dieser Festlegung. Tatsächlich ist sie, wie Sie noch merken werden, nicht immer ganz leicht zu durchblicken.



nannten Instanzvariablen und einer Menge von Instanzmethodendefinitionen. Lediglich indizierte Instanzvariablen sind nicht vorgesehen und der Klassenname kann wie gesagt nicht frei angegeben werden. Dem Schema aus Abschnitt 7.2 folgend sähe eine Metaklassendefinition wie folgt aus:

Klasse	<Klassenname> class
benannte Instanzvariablen	<Liste von Instanzvariablennamen>
Instanzmethoden	<Liste von Methodendefinitionen>

Im konkreten Fall der zweiten Implementierung von Stack oben fände man beispielsweise die folgenden Einträge:

Klasse	Stack class
benannte Instanzvariablen	
Instanzmethoden	
301 new	
302	"liefert neuen Stack mit Platz für 100 Elemente"
303 ..	

Für die Implementierung der Methode **new** fehlt uns noch etwas; sie wird im nächsten Abschnitt nachgeliefert. Hier ist wichtig, dass Sie verstehen, dass **new** eine Instanzmethode der Metaklasse **Stack class** ist und damit das Verhalten der Klasse **Stack** bestimmt und nicht ihrer Objekte.

Aufgrund der bestehenden 1:1-Beziehung zwischen Klassen und Metaklassen werden diese in SMALLTALK nicht getrennt voneinander definiert, sondern in einem gemeinsamen Schema. Jede Klassendefinition verfügt demnach neben den Abschnitten zur Deklaration der Instanzvariablen und zur Definition der Methoden auch über zwei Abschnitte für die entsprechenden Angaben zur ihrer Metaklasse, die Angabe der sog. **Klassenvariablen** und **-methoden**: Es sind dies die Variablen bzw. Methoden, die den Klassen als Instanzen ihrer Metaklassen zugeordnet sind. Das Schema

**Schema einer
Klassendefinition mit
integrierter Meta-
klassendefinition**

Klasse	<Klassenname>
Klassenvariablen	<Liste von Klassenvariablennamen>
Klassenmethoden	<Liste von Methodendefinitionen>
benannte Instanzvariablen	<Liste von Instanzvariablennamen>
indizierte Instanzvariablen	<ja/nein>
Instanzmethoden	



besorgt also nicht nur die Definition der genannten Klasse, sondern gleichzeitig auch die ihrer Metaklasse; es ersetzt damit die zwei zuvor präsentierten getrennten Schemata. Klassenvariablen sind übrigens relativ zu den Instanzen der Klassen global; sie beginnen deswegen mit einem Großbuchstaben. Klassenmethoden schreibt man jedoch wie Instanzmethoden klein. Beachten Sie, dass Klassenvariablen nur einmal pro Klasse angelegt werden — sie sind also für alle Instanzen einer Klasse dieselben.³³

Ein Beispiel für eine Klassenvariable ist **DependentsFields** in der Klasse

Object (zu ihrer Verwendung s. Abschnitt 14.3), eins für eine Klassenmethode ist **pi** in der Klasse **Float**:

304 **pi**
305 ^ Pi

Sie retourniert (den Inhalt der) Klassenvariable **Pi** und ist, da sie eine Klassenmethode ist, allen Instanzen der Klasse **Float** zugeordnet. Dazu, wie der Wert in **Pi** hineinkommt, s. Abschnitt 8.2.

Wir sehen also, dass die Bezeichnungen *Klassenvariable* bzw. *-methode* und *Instanzvariable* bzw. *-methode* eigentlich nur relative Bedeutung haben, da es sich in beiden Fällen um Variablen und Methoden handelt, die Objekten zugeordnet sind. Da man von Instanzen einer Klasse aus aber auch häufiger auf die Variablen und Methoden ihrer Klassen zugreift, ist es guter Brauch (und vermeidet umständliche Formulierungen), stets die langen Bezeichnungen zu führen. Zudem gibt es neben Instanz- und Klassenvariablen ja auch noch andere Variablentypen (*formale Parameter* und *temporäre Variablen*), so dass die Verwendung von „Variable“ allein meist mehrdeutig wäre. Lediglich bei Methoden hat es sich eingebürgert, anstelle von Instanzmethoden nur von Methoden zu sprechen. Wenn der Kontext nichts anderes nahelegt, können Sie dann immer davon ausgehen, dass Instanzmethoden gemeint sind.

Beispiel

Terminologisches

8.1 Konstruktoren

Mit Hilfe von Metaklassen lassen sich nun in SMALLTALK auf natürliche Art und Weise sog. **Konstruktoren** definieren. Ein Konstruktor ist eine Methode, die, auf einer Klasse aufgerufen, eine neue Instanz dieser Klasse zurückgibt (es handelt sich also aus Sicht der Instanzen der Klasse um eine Klassenmethode). Wir haben bereits zwei Konstruktoren von SMALLTALK

³³ Diejenigen unter Ihnen, die eben die Konstruktoren schon kannten, kennen vermutlich auch statisch deklarierte Variablen und Methoden aus JAVA etc.; diese entsprechen im wesentlichen den Klassenvariablen und -methoden SMALLTALKS (auch wenn in JAVA et al. Klassen keine Instanzen von Metaklassen sind).



kennengelernt: Sie werden über die Selektoren **new** (für Objekte ohne indizierte Instanzvariablen) und **new:** (für Objekte mit indizierten Instanzvariablen) aufgerufen.

Da Klassen selbst Objekte sind, sind **new** und **new:** Instanzmethoden der Klassen. Sie sind in Squeak als

```
306 new
307   ^ self basicNew initialize
308 new: sizeRequested
309   ^ (self basicNew: sizeRequested) initialize
```

implementiert. Dabei sind **basicNew** und **basicNew:** ebenfalls Instanzmethoden der Klasse, deren Implementierung allerdings *primitiv* ist (s. Abschnitt 4.3.7 in Kurseinheit 1). Sie geben eine neue Instanz (ein neues Objekt) der Klasse, auf der sie aufgerufen wurden, zurück. Da durch **basicNew** und **basicNew:** alle Instanzvariablen der erzeugten Objekte den Wert **nil** zugewiesen bekommen, wird auf den neuen Objekten, bevor sie (mittels **^**) zurückgegeben werden, noch die Methode **initialize** aufgerufen, die eine Instanzmethode des neuen Objekts ist und die die Instanzvariablen je nach Klasse, in der die Methode definiert ist, anders belegt.

8.2 Initialisierung

Konstruktoren sind in SMALLTALK also Klassenmethoden, die neue Instanzen der jeweiligen Klasse zurückliefern. Dabei haben zunächst alle Instanzvariablen nach der Erzeugung einer Instanz den Wert **nil**. Sollen diese Instanzvariablen mit sinnvollen Anfangswerten belegt werden, müssen ihnen diese explizit zugewiesen werden. Man spricht dann von einer **Initialisierung** der Instanz.

Nun sollen nicht immer alle Instanzen einer Klasse gleich initialisiert werden. Es ist daher möglich, für eine Klasse mehrere alternative Konstruktoren (als Klassenmethoden) zu definieren, die die neuen Objekte jeweils unterschiedlich initialisieren. Zwei Beispiele für die Klasse **Time** sind mit

```
310 midnight
311   ^ self seconds: 0
312 noon
313   ^ self seconds: (SecondsInDay / 2)
```

gegeben, die jeweils die Klassenmethode (den Konstruktor) **seconds:** auf **Time** (vertreten durch **self**) aufrufen, die wiederum mittels **basicNew** eine Instanz von **Time** erzeugt und anschließend initialisiert:

```
314 seconds: seconds
315   ^ self basicNew ticks: (Duration seconds: seconds) ticks
```



Dabei ist **ticks**: eine Instanzmethode der Klasse **Time**, die auf der (mit **basicNew**) frisch erzeugten Instanz aufgerufen wird und diese initialisiert:

```
316 ticks: anArray
317     "ticks is an Array: { days. seconds. nanoSeconds }"
318     seconds := anArray at: 2.
319     nanos := anArray at: 3
```

Parameter der Initialisierung ist hierbei (**Duration seconds: seconds**) **ticks**, wobei **Duration** eine Klasse und **seconds**: ein Konstruktor dieser Klasse ist.

Da die Instanzvariablen eines Objekts nur für die Instanzen des Objekts selbst zugreifbar sind, kann auch eine Klassenmethode wie **new** nicht auf sie zugreifen. Die Initialisierung muss daher von Instanzmethoden wie **ticks**: vorgenommen werden, die jedoch nicht der Initialisierung vorbehalten sind, sondern jederzeit auf Instanzen der Klasse aufgerufen werden können. Das ist immer dann ein Problem, wenn auch Instanzvariablen initialisiert werden müssen, deren Existenz nach außen verborgen werden soll (s. Abschnitt 4.3.4) und die deswegen nicht direkt über Zugriffsmethoden gesetzt werden können sollen. Aus diesem Grund sehen **new** und **new**: standardmäßig den Aufruf der Methode **initialize** vor (s. Zeilen 307 und 309 oben), in der alle Initialisierungen vorgenommen werden können, ohne dass etwas über den Aufbau der Instanzen nach außen verraten würde. In anderen Sprachen wie beispielsweise C++, JAVA oder C# sind Konuktoren daher auch keine Klassenmethoden, sondern haben eine Art Zwitterstatus: Sie werden auf einer Klasse aufgerufen, werden aber wie Instanzmethoden auf der neuen Instanz ausgeführt und können somit auch auf die Instanzvariablen der neu erzeugten Instanz zugreifen. Man beachte jedoch, dass die Instanzmethode **ticks**: kein *Implementationsgeheimnis* preisgibt: Dass Objekte der Klasse **Time** die Zeit in Sekunden und Nanosekunden speichern ist an der Methode **ticks**: nicht zu erkennen.

Vor diesem Hintergrund können wir das Beispiel der zweiten Implementierung von **Stack** aus Abschnitt 7.2 wieder aufgreifen und die noch fehlende Initialisierung der Variablen **stackcontent** und **stackcounter** nachliefern:

Initialisierung von Stacks

Klasse	Stack
Klassenmethoden	
320 new	"liefert neue, gebrauchsfertige Instanz von Stack"
321	^ self basicNew initialize
benannte Instanzvariablen	stackcontent stackpointer
indizierte Instanzvariablen	nein
Instanzmethoden	
323 initialize	"setzt Anfangswerte"
324	stackcontent := Array new: 100.
325	stackpointer := 0



```

327     ^ self "kann entfallen"

328 push: anElement
329     "legt neues Element auf Stapel"
330     stackpointer = stackcontent size
331     ifTrue: [self error: 'Stack leider voll']
332     ifFalse: [ stackpointer := stackpointer + 1.
333                 stackcontent at: stackpointer put: anElement]

334 pop
335     "entfernt oberstes Element vom Stapel"
336     stackpointer = 0
337     ifTrue: [self error: 'Stack leider leer']
338     ifFalse: [ stackpointer := stackpointer - 1]

339 top
340     "liefert oberstes Element des Stacks"
341     stackpointer = 0
342     ifTrue: [self error: 'Stack leider leer']
343     ifFalse: [^ stackcontent at: stackpointer]

```

Man beachte, dass das Zwischenobjekt eine Instanz der Klasse `Array` ist, die hier (in Zeile 325) nicht wie noch in Kurseinheit 1 notwendig durch ein Literal, sondern durch eine explizite, programmatische Instantiierung (mittels `new:`) erzeugt wurde.

Alternativ zu obiger Konstruktion kann die Initialisierung von Instanzvariablen auch zu einem späteren Zeitpunkt nach der Instantiierung durchgeführt werden. Man spricht dann von einer **Lazy initialization** (lazy oder faul deswegen, weil man die Initialisierung solange hinausschiebt, wie irgend möglich). Dazu muss jedoch vor jedem lesenden Zugriff auf die (faul initialisierte) Instanzvariable geprüft werden, ob der Wert der Variable immer noch `nil` ist — falls ja, muss er durch den gewünschten Anfangswert (der sonst in der Standardinitialisierungsmethode zu finden wäre) ersetzt werden. Um nicht jeden lesenden Zugriff auf die Variable im Programm mit einer entsprechenden Abfrage versehen zu müssen, empfiehlt es sich bei Verwendung von *Lazy initialization*, alle, also auch klasseninterne, Zugriffe auf Instanzvariablen über einen entsprechenden *Getter* durchzuführen, der den Inhalt der Variable vor seiner Preisgabe prüft und ggf. erst setzt. Statt

| Lazy initialization |

```

344 push: anElement
345     "legt neues Element auf Stapel"
346     stackpointer isNil ifTrue: [stackpointer := 0].
347     stackcontent isNil ifTrue: [stackcontent := Array new: 100].
348     stackpointer = stackcontent size
349     ifTrue: ...

350 pop
351     "entfernt oberstes Element vom Stapel"
352     stackpointer isNil ifTrue: [stackpointer := 0].
353     stackpointer = 0
354     ifTrue: ...

355 top
356     "liefert oberstes Element des Stacks"

```



```
357 stackpointer isNil ifTrue: [stackpointer := 0].  
358 stackpointer = 0  
359 ifTrue: ...
```

wo bei jeder Verwendung ggf. faul initialisiert wird, würde man also

```
360 stackpointer  
361     "lazy: liefert den ggf. zuvor initialisierten Stack pointer"  
362     stackpointer isNil ifTrue: [stackpointer := 0].  
363     ^ stackpointer  
  
364 stackcontent  
365     "lazy: liefert den ggf. zuvor initialisierten Stack content"  
366     stackcontent isNil ifTrue: [stackcontent := Array new: 100].  
367     ^ stackcontent  
  
368 push: anElement  
369     "legt neues Element auf Stapel"  
370     self stackpointer = self stackcontent size  
371     ifTrue: ...  
  
372 pop  
373     "entfernt oberstes Element vom Stapel"  
374     self stackpointer = 0  
375     ifTrue: ...  
  
376 top  
377     "liefert oberstes Element des Stapels"  
378     self stackpointer = 0  
379     ifTrue: ...
```

schreiben (man beachte das **self** vor **stackpointer** und **stackcontent** — hier wird jeweils ein Getter aufgerufen).

Wie man sieht, ist die Programmiererin bei der Lazy initialization überhaupt nicht faul — sie muss sogar einiges mehr an Code schreiben, als bei einer Standardinitialisierung notwendig wäre. Das laufende Programm spart sich jedoch den Preis der Initialisierung, wenn diese nie notwendig wird, wenn also im Programmablauf der Wert der zu initialisierenden Variable nie oder erst nach einer anderen Zuweisung abgefragt wird (was im Beispiel vom Stack freilich nicht der Fall ist). Sie lohnt sich also immer dann, wenn die Initialisierung aufwendig und die Abfrage des Anfangswertes selten ist. Ein weiterer Vorteil der Lazy initialization ist, dass die Initialisierung nie vergessen werden kann; dies ist insbesondere dann wertvoll, wenn die Initialisierung nicht wie oben beschrieben vom Konstruktor selbst, sondern von einer separaten Methode durchgeführt wird und den Benutzerinnen der entsprechenden Klasse vielleicht nicht klar ist, dass sie nach dem Konstruktor auch noch die Initialisierungsmethode aufrufen müssen. Konstruktoren, die wie in Zeilen 307 und 309 oben) implementiert wurden, suchen das zu verhindern, indem sie die Initialisierungsmethode selbst aufrufen; manchmal kann der Konstruktor doch nur schlecht geändert werden (s. z. B. Abschnitt 10.3) und man wird auch nicht verhindern können, dass, anstelle von **new**, **basicNew** direkt und ohne **initialize** aufgerufen wird.

**Vor- und Nachteile
einer Lazy
Initialization**

Selbsttestaufgabe 8.1

Begründen Sie, warum eine Kapselung der Lazy initialization durch eine Zugriffsmethode dem Sinn der Standardinitialisierung per `initialize` möglicherweise entgegensteht.

Nachdem nun hinlänglich klargeworden sein sollte, welche Möglichkeiten es zur Initialisierung von Instanzvariablen gibt, bleibt noch die Frage nach der Initialisierung von Klassenvariablen. Klassenvariablen werden nämlich, genau wie Instanzvariablen, standardmäßig zu `nil` initialisiert und soll eine Klassenvariable einen anderen Anfangswert haben (z. B. weil es sich dabei um eine Konstante handelt, die für alle Instanzen der Klasse eine Rolle spielt), dann muss ihr dieser Wert explizit zugewiesen werden. Da Klassen ja Instanzen ihrer Metaklassen sind, diese Metaklassen aber automatisch mit der Erzeugung der Klassen angelegt werden und das Klassendefinitionsschema keine Möglichkeit vorsieht, einen Konstruktor für die Metaklasse vorzugeben, muss eine spezielle Klassenmethode (häufig ebenfalls „`initialize`“ genannt) für die Initialisierung der Klassenvariablen vorgesehen werden. Diese ist dann nach Anlegen der Klasse einmalig aufzurufen. Da das aber leicht vergessen werden kann, ist *Lazy initialization* für Klassenvariablen eine sinnvolle Alternative. Allerdings stellt sich hier wieder das Problem des direkten Zugriffs auf die (Klassen-)Variable (aus dem Kontext der Klasse selbst und ihrer Instanzen), der in SMALLTALK nicht unterbunden werden kann (vgl. Selbsttestaufgabe 8.1).

Initialisierung von Klassenvariablen

Selbsttestaufgabe 8.2

Schreiben Sie eine Methode `new`, die dafür sorgt, dass alle mit ihr erzeugten Instanzen in einer Klassenvariable `MeineInstanzen` gespeichert werden.

8.3 Factory-Methoden

Da in SMALLTALK Konstruktoren ganz normale Klassenmethoden sind, sind sie an keine besonderen Konventionen gebunden. Sie müssen also insbesondere nicht ein neues Objekt genau der Klasse, der sie angehören, zurückgeben. Dies nutzen die sog. Factory-Methoden aus.

Eine **Factory-Methode** ist eine Methode, die wie ein Konstruktor eine neue Instanz liefert, die aber die Klasse der Instanz von anderen Faktoren als nur der Klasse, zu der die Methode gehört, abhängig macht. Zum Beispiel könnte eine Klasse `Number` eine (Klassen-)Methode `fromString:` vorsehen, die anhand eines zu analysierenden Strings entweder eine Instanz der Klasse `Integer` oder eine

Erzeugung von Instanzen beliebiger Klassen



der Klasse **Float** zurückgibt. Die Implementierung solcher Factory-Methoden ist in SMALLTALK leicht möglich; sie unterscheiden sich formal auch überhaupt nicht von Konstruktoren — es sind einfach alles Klassenmethoden.³⁴

Folgende Klassenmethode der Klasse **Number** (für beliebige Zahlen) ist eine Factory-Methode:

```
Klasse | Number
Klassenmethoden |  

380 fromString: aString
381     "Factory für Zahlen"
382     (aString includes: $.)
383     ifTrue: [^ aString asFloat]
384     ifFalse: [^ aString asInteger]
```

Wenn der Parameter **aString** einen Dezimalpunkt enthält, wird eine neue Fließkommazahl zurückgegeben (mittels der Methode **asFloat**, die, in der Klasse **String** implementiert, eine Instanz der Klasse **Float** zurückliefert), sonst eine Ganzzahl.

8.4 Erzeugung von Klassen in SMALLTALK

Da Instanzen, für die es keine literale Repräsentation gibt, in SMALLTALK grundsätzlich über Konstruktoren erzeugt werden und jede Klasse Instanz ihrer Metaklasse ist, kann man sich fragen, wie in SMALLTALK eigentlich Klassen erzeugt werden. Die Tatsache, dass es einen Browser mit einer entsprechenden Funktion gibt, reicht als Erklärung hierfür nicht aus. Andererseits kann es auch keine Lösung sein, einfach **new** o. ä. an die Metaklasse der Klasse zu senden, da diese ja zunächst noch gar nicht existiert. Es stellt sich hier tatsächlich die sprichwörtliche Frage nach der Henne und dem Ei, genauer, wer von beiden zuerst existieren muss.

Dieses Dilemma wird von SMALLTALK intern gelöst. Eine Klasse wird in SMALLTALK nämlich erzeugt, indem man einer anderen Klasse eine entsprechende Nachricht schickt. Der Protokolleintrag der dazugehörigen Methode (je nach System in der Klasse **Class** oder **Behavior** zu finden) für Klassen ohne indizierte Instanzvariablen sieht in SQUEAK wie folgt aus:

Verwendung der Klassen Class und Behavior zur Erzeugung einer Klasse

```
385 subclass: t instanceVariableNames: f classVariableNames: d
      poolDictionaries: s category: cat
      "This is the standard initialization message for creating a new
       class as a subclass of an existing class (the receiver)."
388   ^(ClassBuilder new)
389   superclass: self
390   subclass: t
```

³⁴ Das erklärt vermutlich auch, warum der Begriff des Konstruktors in SMALLTALK wenig gebräuchlich ist.



```
391     instanceVariableNames: f  
392     classVariableNames: d  
393     poolDictionaries: s  
394     category: cat
```

Durch Ausführung dieser Methode wird eine neue Klasse und zugleich ihre Metaklasse angelegt. Dabei kann man dem Kommentar entnehmen, dass die neue Klasse *Subklasse* des Empfängers (einer Klasse) werden soll. Was das heißt, wird Gegenstand von Kapitel 11 sein. Hier wenden wir uns lieber der Frage zu, von welchen Klassen die miterzeugten Metaklassen Instanzen sind.

8.5 Die Metaklassenleiter SMALLTALKS

Es müssen nach der SMALLTALK-Philosophie auch Metaklassen (als Objekte) Instanzen von Klassen sein. Da es aber nicht mehr sinnvoll erscheint, jeder Metaklasse eigene Instanzvariablen und Methoden zu geben, ist es nicht notwendig, dass jede Metaklasse (als Klasse) ihre eigene Meta-Metaklasse (als Metaklasse der Klasse) hat. Vielmehr reicht es für die Praxis aus, eine gemeinsame Meta-Metaklasse, von der alle Metaklassen Instanzen sind, vorzusehen. SMALLTALKS Benennungspraxis, nach der jede Klasse so heißt, dass ihre Instanzen als Subjekt den Satz „<eine Instanz> ist ein <Klassenname>“ korrekt ergänzen, folgend heißt diese Klasse **Metaclass** (da eben alle ihre Instanzen Metaklassen sind).

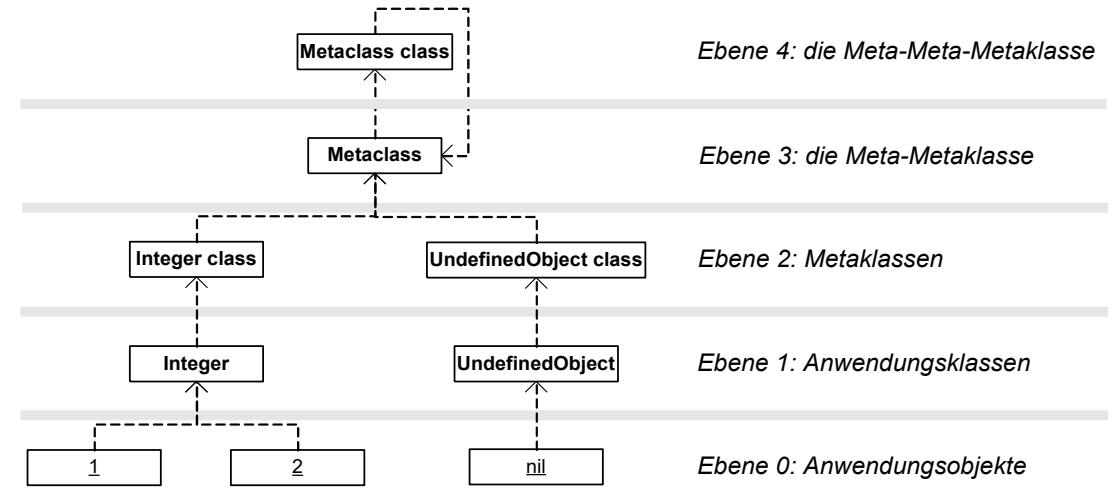
Es ergibt sich sofort die Frage, von welcher Klasse die Klasse **Metaclass** eine Instanz ist — tatsächlich muss ja nach der Philosophie SMALLTALKS, nach der Klassen Objekte und jedes Objekt Instanz einer Klasse ist, auch **Metaclass** Instanz einer Klasse sein. Um dieses Spiel nicht bis ins Unendliche fortsetzen zu müssen, hat man in SMALLTALK zu einem einfachen Trick gegriffen: Man betrachtet die Klasse von **Metaclass**, also **Metaclass class**, selbst nur als einfache Metaklasse (obwohl sie ja eigentlich eine Meta-Meta-Metaklasse ist), die, genau wie alle anderen Metaklassen, Instanz von **Metaclass** sein muss. Es gilt also für **Metaclass**

Metaklasse der Metaklassen

```
395 Metaclass class class == Metaclass
```

Nachfolgendes Diagramm veranschaulicht die Zusammenhänge. Man beachte, dass alle Objekte bis auf die der Ebene 0 gleichzeitig Klassen und Instanzen sind. Der gestrichelte Pfeil bezeichnet übrigens die Ist-eine-Instanz-von-Beziehung (in UML-Notation).





Die theoretisch ambitionierte Leserin wird sofort bemerken, dass der Kunstgriff der Terminierung der Ist-eine-Instanz-von-Beziehung es verbietet, Klassen als Mengen von Objekten und deren Instanzen als Elemente dieser Mengen im Sinne von Abschnitt 7.3 zu interpretieren: Sonst wäre nämlich die zu **Metaclass** gehörende Menge von Objekten indirekt ein Element von sich selbst, was schlechterdings unmöglich ist. Außer einem etwas faden Beigeschmack hat das jedoch keine praktischen Auswirkungen.

Verlust der mengentheoretischen Interpretation

Wir haben es also in SMALLTALK mit einem mehrstufigen Zusammenspiel von Klassen und Instanzen zu tun. Auf der untersten Stufe, der Ebene 0, stehen konkrete Objekte, die nicht instanziierbar sind. Diese Objekte repräsentieren in der Regel Dinge aus dem Anwendungsbereich eines Programms, also zum Beispiel konkrete Personen, Dokumente, Adressen etc. Eine Stufe darüber, auf Ebene 1, stehen die Klassen, die die Definition (Instanzvariablen und -methoden) dieser Objekte liefern und anhand derer die Objekte der Ebene 0 (per Instanziierung) erzeugt werden. Diese Klassen repräsentieren die Objekte der Ebene 0 in ihrer Gesamtheit; sie repräsentieren die Konzepte oder Allgemeinbegriffe des Anwendungsbereichs. Zu jeder Klasse der Ebene 1 werden im Laufe des Programms in der Regel mehrere Objekte der Ebene 0 erzeugt — es besteht also eine 1:n-Beziehung zwischen ihnen.

stufenweise Reduktion der Vielfalt durch Klassifikation

Nun sind auch die Klassen der Ebene 1 Objekte und damit selbst Instanzen von Klassen, die eine Stufe höher, also auf Ebene 2 stehen. Die Klassen der Ebene 2, die Metaklassen, geben die Definition der Klassen vor. Da es nicht sinnvoll ist, von Klassen der Ebene 1 mehrere Exemplare zu haben, die, analog zu den Objekten der Ebene 0, alle über die gleiche Definition verfügen, hat jede Metaklasse genau eine Instanz. Es besteht also eine 1:1-Beziehung zwischen Metaklassen und ihren Instanzen, den Klassen der Ebene 1, die die Objekte der Anwendung beschreiben.

Auf Ebene 3 bekommen alle Metaklassen eine gemeinsame Klasse spendiert, von der sie eine Instanz sind, nämlich die Klasse **Metaclass**. Man beachte, dass hier wieder eine 1:n-Beziehung vorliegt. Anders als auf Ebene 2, auf der man für die unterschiedlichen Konzepte



einer Anwendung jeweils eine Klasse vorfindet, hat man hier, auf Ebene 3, die Vielfalt auf genau eine Klasse verdichtet. Diese hat dann wieder genau eine Metaklasse.

Das nachfolgende Diagramm zeigt noch einmal die Reduktion durch die ersten vier Stufen. Eine ähnliche Verdichtung über vier Ebenen findet man übrigens auch beim Information Resource Dictionary System (IRDS) der ISO.

Link



8.6 Praktische Bedeutung der Metaklassen für die Programmierung

Dadurch, dass in SMALLTALK Klassen Instanzen von Metaklassen sind, die selbst Instanzen einer weiteren Klasse und diese alle zusammen Objekte sind, ist jedes SMALLTALK-Programm, ja das ganze SMALLTALK-System, nichts weiter als ein Objektgeflecht (sieht man einmal von den primitiven Methoden ab). SMALLTALK ist damit nicht nur ein Programmiersystem, sondern auch ein **Metaprogrammiersystem** in der Tradition funktionaler und logischer Programmiersprachen wie LISP und PROLOG. In der imperativen und objektorientierten Programmiersprachenlandschaft sucht diese Mächtigkeit bis heute ihresgleichen.

SMALLTALK als
Metaprogrammier-
system

Für Sie als Programmiererin, die nicht gleich eine neue Sprache erschaffen will, sind Ebene 2 und 3 sind vor allem dann interessant, wenn Sie sich im Inneren von SMALLTALK umsehen oder es vielleicht sogar selbst verändern wollen. Wenn Sie z. B. erreichen wollen, dass beim Anlegen einer neuen Klasse für alle benannten Instanzvariablen dieser Klasse automatisch Zugriffsmethoden wie in Abschnitt 4.3.4 definiert werden, dann ist dies leicht möglich, indem Sie an entsprechender Stelle (z. B. in der Klasse **Class** bzw. **Behavior**, die auf der Ebene der Metaklassen steht und die für das Anlegen neuer Klassen zuständig ist) eine neue Methode zur Klassendefinition einfügen, die die bereits existierenden um die automatische Erzeugung der Zugriffsmethoden ergänzt.

Selbsttestaufgabe 8.3

Ergänzen Sie die Klasse **Class** um eine Methode zur Anlage neuer Klassen, die für ausgewählte Instanzvariablen automatisch Zugriffsmethoden (Accessoren; einen Setter und einen Getter) zum Methodenkatalog hinzufügt. Teilen sie dazu die bei einer Klassendefinition angegebene Liste der Instanzvariablen in zwei auf, von denen die eine ohne, die andere mit Accessoren angelegt wird.

Im Programmieralltag werden Sie das aber nicht tun. Vielmehr beschränkt sich Ihre Tätigkeit da auf das Anlegen und Ändern einfacher Klassen, also solcher, deren Instanzen selbst keine Klassen sind. Die dazu notwendigen Metaklassen erzeugt SMALLTALK automatisch selbst — im Klassenbrowser erscheinen sie nur über die Unterscheidung zwischen Instanz- und Klassenvariablen bzw. -methoden.

Programmieralltag



9 Generalisierung und Spezialisierung

Es gibt in SMALLTALK also eine Hierarchie, die auf dem Konzept der Klassifikation aufbaut. Aufgrund praktischer Überlegungen ist diese Hierarchie beschränkt; sie ist mit der Sprachdefinition festgelegt und stellt gewissermaßen einen Teil derselben dar. Konzeptionell ist diese Hierarchie eine *Abstraktionshierarchie*: Von den konkreten Objekten der Ebene 0 geht es über die Allgemeinbegriffe oder Konzepte der Ebene 1 zu den Definitionen dieser Konzepte auf Ebene 2 hin zur Fassung von Definitionen allgemein auf Ebene 3. Mit jeder Stufe mit Ausnahme der mittleren wird die Zahl der Objekte, die unter die darin angesiedelten Konzepte fallen, drastisch reduziert: von Ebene 0 auf Ebene 1 von theoretisch unendlich vielen Objekten einer Anwendung zur Zahl der Anwendungsklassen, von Ebene 2 auf Ebene 3 von der Zahl der Anwendungsklassen zu einer Klasse **Metaclass**. Als Programmiererin bewegen Sie sich jedoch vor allem auf Ebene 1: Sie definieren Anwendungsklassen, von denen zur Laufzeit des Programms die Anwendungsobjekte erzeugt werden. Direkt nutzen Sie also nur eine Abstraktionsstufe für die Programmierung.

9.1 Generalisierung

Nun entspricht, wie eingangs (in Abschnitt 7.1) erwähnt, die Klassifikation sprachlich der *Ist-ein-Abstraktionsbeziehung* zwischen Individuen und ihren Klassen: „Peter ist ein Mensch“, „SMALLTALK ist eine Programmiersprache“ usw. sind alles Beispiele für eine Art der Abstraktion, bei der man von einem Individuum zu seinem Allgemeinbegriff übergeht. Es gibt aber noch eine zweite Form der Ist-ein-Abstraktion, die sich von der ersten fundamental unterscheidet, die aber ebenfalls eine charakteristische Rolle in der objektorientierten Programmierung spielt: die **Generalisierung**. Sprachlich offenbart sich diese in Sätzen wie „ein Mensch ist ein Säugetier“, „ein Säugetier ist ein Lebewesen“ oder „eine Programmiersprache ist ein Werkzeug“. Der Unterschied zur ersten Form der Abstraktion liegt dabei offensichtlich darin, dass hier zwei Allgemeinbegriffe und nicht ein Individuum und ein Allgemeinbegriff miteinander ins Verhältnis gesetzt werden. Ein weiterer, etwas subtilerer, aber sehr wesentlicher Unterschied ist der, dass die Klassifikation nicht transitiv ist, die Generalisierung hingegen schon. So folgt aus „ein Mensch ist ein Säugetier“ und „ein Säugetier ist ein Lebewesen“ wohl „ein Mensch ist ein Lebewesen“, aber aus „Peter ist ein Mensch“ und „Mensch ist eine Art“ nicht „Peter ist eine Art“.

Selbsttestaufgabe 9.1

Ordnen Sie der nachfolgenden Sequenz von Ist-ein-Sätzen die jeweilige Form der Abstraktion zu:

1. Clyde ist ein Elefant.
2. Elefant ist ein Säugetier.
3. Säugetier ist ein Wirbeltier.
4. Wirbeltier ist ein Stamm.
5. Stamm ist ein Taxon.



6. Elefant ist eine Spezies.
7. Spezies ist ein Taxon.

Bilden Sie alle daraus ableitbaren Ist-ein-Sätze und bestimmen Sie die längste Ableitung.

Beim *Vorgang der Generalisierung* werden mehrere Klassen, deren Definitionen inhaltlich verwandt sind, zusammengefasst, wobei das *Ergebnis der Generalisierung*, ebenfalls Generalisierung genannt, eine Klasse ist, die nur diejenigen Elemente der Definitionen der generalisierten Klassen enthält, die allen gemeinsam sind. So lässt sich beispielsweise aus den beiden ähnlichen, aber nicht gleichen Klassen **Mensch**

Vorgang und Ergebnis der Generalisierung

Klasse	Mensch
benannte Instanzvariablen	linkesBein rechtesBein aufenthaltsort verstand
indizierte Instanzvariablen	nein
Instanzmethoden	
396 laufeNach: neuerOrt	"bewegt Empfänger per pedes an neuen Ort"
397	...
398	
399 rechne: eineAufgabe	"löse die Aufgabe mittels Verstand"
400	...
401	

und **Vogel**

Klasse	Vogel
benannte Instanzvariablen	linkesBein rechtesBein aufenthaltsort flügel
indizierte Instanzvariablen	nein
Instanzmethoden	
402 laufeNach: neuerOrt	"bewegt Empfänger per pedes an neuen Ort"
403	...
404	
405 fliegeAn: neuenOrt	"bewegt Empfänger durch Flügel an neuen Ort"
406	...
407	

per Vorgang Generalisierung die Klasse **Zweibeiner**

Klasse	Zweibeiner
benannte Instanzvariablen	linkesBein rechtesBein aufenthaltsort
indizierte Instanzvariablen	nein



```
408 laufeNach: neuerOrt
409     "bewegt Empfänger per pedes an neuen Ort"
410     ...
```

herausfaktorisieren, deren Definition, als Ergebnis der Generalisierung der Klassen **Mensch** und **Vogel**, genau die gemeinsamen Eigenschaften (Instanzvariablen und Methoden) entält.

Da die Eigenschaften, die einer Generalisierung als Klasse zugeordnet sind, per Definition automatisch auch für alle Klassen, von denen die Generalisierung abstrahiert, gelten (denn das war ja die Bedingung für die Konstruktion der Generalisierung), brauchen diese die Eigenschaften nicht zu wiederholen, sondern stattdessen nur noch ihre Generalisierung anzugeben. Diese Klassen müssen dann nur noch die Unterschiede, die sie von **Zweibeiner** sowie von einander unterscheiden, definieren:

Ökonomie der Generalisierung

Klasse	Mensch
Generalisierung	Zweibeiner
benannte Instanzvariablen	verstand
indizierte Instanzvariablen	nein
Instanzmethoden	

```
411 rechne: eineAufgabe
412 ...
```

bzw.

Klasse	Vogel
Generalisierung	Zweibeiner
benannte Instanzvariablen	flügel
indizierte Instanzvariablen	nein
Instanzmethoden	

```
413 fliegeAn: neuenOrt
414 ...
```

Diese zweite Form der Abstraktion, die Generalisierung, ist also genau wie die Klassifikation Bestandteil der klassenbasierten objektorientierten Programmierung. Anders als bei der Klassifikation ist bei der Generalisierung aber die Höhe der Abstraktionshierarchie nicht durch praktische Überlegungen beschränkt, sondern kann von der Programmiererin nach Belieben angelegt werden. Sprachphilosophisch sind Generalisierungen nämlich genau wie Klassen **Allgemeinbegriffe**; sie sind nur noch allgemeiner. Generalisierungen können somit selbst Generalisierungen haben und so weiter; wie sich das für eine Abstraktionshierarchie gehört, werden die Definitionen, die *Intensionen*, dabei immer knapper. Gleichzeitig wächst

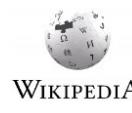


jedoch die *Extension* (das bereits in Abschnitt 7.1 erwähnte Prinzip vom inversen Zusammenhang der beiden).

Abgeschaut ist das Prinzip der Generalisierung übrigens von *Aristoteles'* Prinzip von *Genus et differentia*, der gemeinsamen Abstammung und den

philosophischer
Vorläufer

Unterschieden: Das Genus ist die nächst allgemeinere Kategorie, unter die die Objekte der zu generalisierenden Klassen (der Spezies) auch fallen, und die Differentia sind die Kriterien, nach denen sich die Objekte aufgrund ihrer Natur, wie sie in den verschiedenen Klassendefinitionen festgelegt (und nicht etwa durch spezielle Werte von Instanzvariablen bestimmt) ist, unterscheiden. So haben eben die Klassen *Mensch* und *Vogel* das gemeinsame Genus *Zweibeiner* als (biologisch nicht ganz korrekte) Generalisierung: In ihr ist festgelegt, dass alle Exemplare von Zweibeinern (und damit auch von Menschen und Vögeln) ein linkes und ein rechtes Bein sowie einen Aufenthaltsort haben. Die Unterschiede (Differentia) sind dann in den jeweiligen Klassen herausgearbeitet. Man beachte, dass Genera keine eigenen Individuen haben, also keine Individuen, die nicht Individuen einer ihrer Spezies wären. So gibt es keine Zweibeiner, die nicht entweder Mensch oder Vogel wären.³⁵



WIKIPEDIA

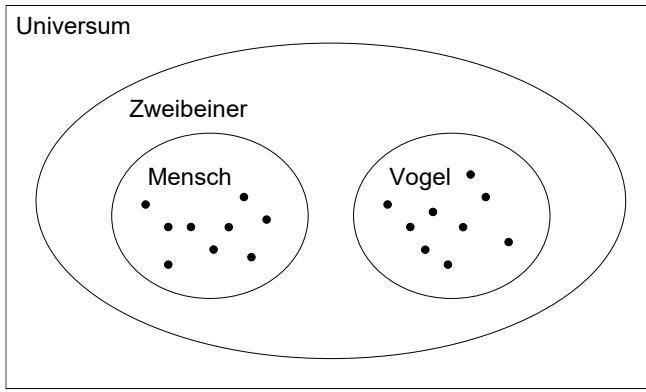
Genau wie die Klassifikation hat das Ordnungsprinzip der Generalisierung eine einfache mengentheoretische Interpretation. Demnach enthält die

mengentheoretische
Interpretation

Menge der Instanzen einer Generalisierung alle Instanzen der Klassen, von denen sie eine Generalisierung ist. Wenn also **Mensch** und **Vogel** Ausgangsklassen einer Generalisierung **Zweibeiner** sind, dann ist die Menge der Instanzen, die **Zweibeiner** repräsentiert (für die **Zweibeiner** den Allgemeinbegriff abgibt) eine Obermenge der Vereinigung der Menge der Instanzen von **Mensch** und **Vogel**. Die Menge der Instanzen von **Zweibeiner** ist eine echte Obermenge, wenn **Zweibeiner** auch noch eigene Instanzen hat (also Instanzen, die nicht Instanzen von **Mensch** und **Vogel** sind; im Kontext der Instanzierung würde man von *direkten Instanzen* sprechen; s. Abschnitt 7.3); sonst ist sie nur eine unechte Obermenge (also genau gleich der Vereinigung). Die nachfolgende Grafik zeigt den Zusammenhang (wobei die schwarzen Punkte die Instanzen und die Ellipsen die Klassen darstellen sollen). Gute Praxis (und hier angedeutet) ist, wenn Generalisierungen keine eigenen, direkten Instanzen haben, also Genera im obigen Sinne sind. Dies ist in der objektorientierten Praxis aber (leider) längst nicht immer selbstverständlich, wie sich im nächsten Kapitel noch zeigen wird (vgl. dazu auch Kapitel 69 in Kurseinheit 7).

³⁵ Biologinnen möchten vielleicht **Mensch** durch **Primate** ersetzen.





Die mengentheoretische Interpretation von Generalisierung als Obermengenbildung legt nahe, dass Instanzen von **Mensch** und **Vogel** (als Elemente der entsprechenden Extensionen) auch Instanzen von **Zweibeiner** sind. Wenn man das so sehen will, dann sollte man aber zur notwendigen Unterscheidung von **indirekten Instanzen** (anstelle von *direkten Instanzen*; s. Abschnitt 7.3) sprechen.

Bei der Generalisierung können also Eigenschaften, die verschiedene, aber ähnliche Klassen unterscheiden, weggelassen („wegabstrahiert“) werden. Das Weglassen ist aber nicht die einzige mögliche Form der Generalisierung: Es können auch Eigenschaften generalisiert werden, wobei dann der Begriff der Generalisierung rekursiv zur Anwendung kommt. Dabei versteht man unter der Generalisierung von Attributen (oder allgemeiner von Instanzvariablen; s. Abschnitt 2.4), dass ihr Wertebereich von einem spezielleren (kleineren) zu einem allgemeineren (größeren) aufgeweitet wird. So würde beispielsweise das Attribut **aufenthaltsort**, das mit (Instanzen der) Klasse **Mensch** assoziiert ist, beim Übergang zur Generalisierung **Zweibeiner** von Punkten auf der Erdoberfläche zu Punkten einschließlich des Luftraums darüber generalisiert, so dass es auch den Wertebereich für Vögel abdeckt. In SMALLTALK gibt es aber keine Möglichkeit, Attributen per Deklaration Wertebereiche zuzuordnen; wie Sie noch sehen werden, erlauben zudem aus gutem Grund die wenigsten Programmiersprachen, die die Möglichkeit der Werteschränkung von Variablen vorsehen, Attributwertebereiche bei der Generalisierung ebenfalls zu generalisieren (die sog. *kovariante Redefinition*; s. dazu auch die Kapitel 25 und Abschnitt 26.3 in Kurseinheit 3).

Generalisierung ohne Weglassen von Eigenschaften

Auch wenn bislang so getan wurde, also sei die Generalisierung etwas in der Natur des betrachteten Gegenstandes liegendes, so gibt es in der Praxis jedoch oftmals verschiedene Gesichtspunkte, nach denen man Generalisierungen durchführen kann. So ist z. B. die Generalisierung von **Vogel** bzw. **Mensch** zu **Zweibeiner** nicht die einzige mögliche (und sicher nicht die einzige sinnvolle). Es könnte also durchaus sein, dass man mehrere, voneinander unabhängige Generalisierungshierarchien konstruieren möchte, in denen durchaus dieselben Klassen auftauchen. In der Praxis verliert man dadurch jedoch die strikte Hierarchieform der Generalisierung (da sich mehrere Hierarchien überlagern), es sei denn, man erlaubt, verschiedene Arten der Generalisierung voneinander zu unterscheiden. Beides bringt jedoch einiges an Komplikationen mit sich, so dass wir hier auf „Mehrfachgeneralisierungen“ nicht eingehen werden.

mehrere Dimensionen der Generalisierung



9.2 Spezialisierung

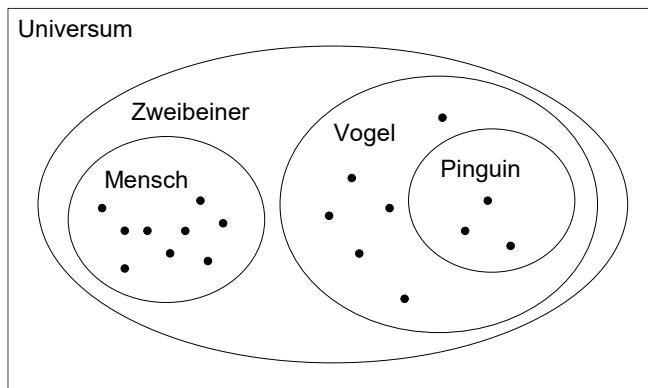
Ähnlich wie bei der Klassifikation kann man das Prinzip der Generalisierung umkehren. Man redet dann von der **Spezialisierung**. Während die Generalisierung Eigenschaften weglässt oder generalisiert (Abstraktion), fügt die Spezialisierung Eigenschaften hinzu oder spezialisiert bereits vorhandene. Man kann also von jeder Klasse sagen, dass sie eine Spezialisierung ihrer Generalisierungen ist (so sie denn welche hat).

Dass eine Generalisierung bereits über Spezialisierungen verfügt, hindert eine nicht daran, neue hinzuzufügen. So ist es beispielsweise im obigen Beispiel von **Zweibeiner** denkbar, dass man im Nachhinein noch **Menschenaffe** als Spezialisierung ergänzt. Als Differentia käme z. B. eine Methode **hangeln** in Frage, die **Mensch** und **Vogel** fehlt. Sie zu ergänzen stellt überhaupt kein Problem dar — ja es ist sogar eine der größten Errungenschaften der objektorientierten Programmierung, dass solche Programmerweiterungen modular, also ohne andere Teile des Programms zu betreffen, immer möglich sind. Mehr dazu in Kapitel 26 in Kurseinheit 3.

Spezialisierung von Generalisierungen

Leider ist es in der Programmierpraxis nicht immer ganz so einfach. Vielmehr findet man häufig Klassen (bzw. Instanzen) vor, die ungefähr das tun, was man möchte, und denen man nur noch ein wenig hinzufügen möchte. Man möchte dann von einer Klasse spezialisieren, die selbst keine Generalisierung im obigen Sinne ist. Um beim obigen Beispiel mit Menschen und Vögeln zu bleiben, könnte man beispielsweise auf den Gedanken kommen, Pinguine als Spezialisierung von Vögeln einzuführen:

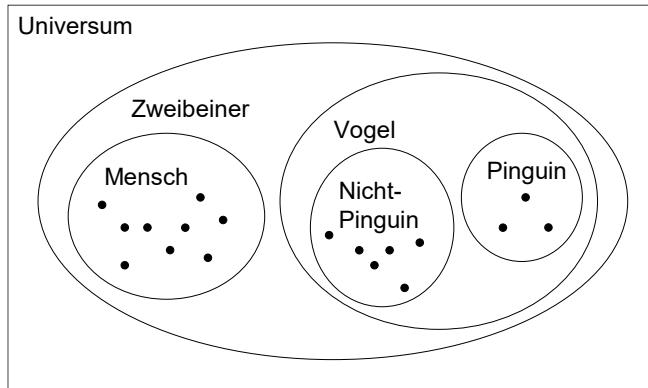
Spezialisierung von Nicht-Generalisierungen



Es ist nun fraglich, ob damit auch **Vogel** zu einer Generalisierung von **Pinguin** wird. Die Tatsache, dass **Vogel** eigene, *direkte* Instanzen hat, spricht schon einmal dagegen (auch wenn der Begriff der Generalisierung landläufig nicht so streng gefasst wird). Weiterhin kann man sich fragen, was man von der Intension von **Pinguin** weglassen müsste, um zur Intension von **Vogel** zu gelangen. Dies könnte z. B. **schwimmeNach**: sein. Spätestens dann fällt einer jedoch auf, dass Pinguine gar nicht fliegen können, also die Intension von **Vogel** die Methode **fliegeNach**: gar nicht enthalten dürfte, wenn **Vogel** eine Generalisierung von **Pinguin** sein sollte. Dieses Problem, das in der Praxis ständig vorkommt, lässt sich auf elegante Weise dadurch lösen, dass man eine Klasse **NichtPinguin** parallel zu **Pinguin**



spezialisiert und alle Eigenschaften, die andere Vögel von Pinguinen unterscheidet (wie z. B. `fliegeAn:`), dort hineinpackt.



Ähnlich wie bei der Generalisierung ist es bei der Spezialisierung auch möglich, dies ohne das Hinzufügen von Eigenschaften zu bewerkstelligen, nämlich durch das Einschränken von Eigenschaften. So kann man z. B. bei der Spezialisierung von **Säugetier** zu **Zweibeiner** den Wertebereich der Instanzvariable `anzahlBeine` von $\{2, 4\}$ (also entweder zwei oder vier) auf $\{2\}$ (also nur noch zwei) eingeschränkt werden. Die sprachlichen Möglichkeiten, dies auf Klassendefinitionsebene auszudrücken, sind allerdings in SMALLTALK (wie bereits im Zusammenhang mit der Generalisierung erwähnt) nicht gegeben; sie kommen erst mit der Typisierung von Variablen (Kapitel 18 in Kurseinheit 3). Die Einschränkung des Wertebereichs per Spezialisierung ist aber in jedem Fall zu unterscheiden von der Instanziierung, im Zuge derer (ggf. über eine *Initialisierung*) einer Instanzvariable eines Objekts ein Element aus dem Wertebereich (wie z. B. 2) zugewiesen wird. Dass im Fall von **Zweibeiner** dafür dann nur noch ein Element als Wert in Frage kommt, spielt dabei keine Rolle.

Spezialisierung ohne Hinzufügen von Eigenschaften

In der objektorientierten Programmierung Instanziierung mit Spezialisierung zu verwechseln ist genau so sträflich wie in der Mathematik Elementsein (\in) mit Enthaltensein (\subseteq).

Vollkommen unvereinbar mit der Spezialisierung ist übrigens, Instanzvariablen oder Methoden wegzunehmen. Dies folgt schon daraus, dass die Umkehrung der Spezialisierung, die Generalisierung, dann nicht aus dem bloßen Weglassen entstanden sein könnte. Die Richtung von Spezialisierung und Generalisierung würde zudem, wenn nach Belieben in beide Richtungen hinzugefügt und wegegenommen werden dürfte, ebenfalls beliebig.

Spezialisierung darf nichts wegnehmen



10 Vererbung und abstrakte Klassen

Generalisierung und Spezialisierung wie oben dargestellt sind eher theoretisch motivierte Konzepte. In der Programmierung geht man jedoch häufig, wie im obigen Beispiel von Pinguinen schon angedeutet, an praktischen Gesichtspunkten orientiert vor. So haben denn auch nicht Generalisierung und Spezialisierung die Entwicklung objektorientierter Programmiersprachen geprägt, sondern *abstrakte Klassen* und *Vererbung*. Diese pragmatische Orientierung ist jedoch nicht ohne Probleme und so werden uns die Überlegungen zu Generalisierung und Spezialisierung spätestens in Kurseinheit 3 wieder begegnen.

10.1 Vererbung

Unter **Vererbung** versteht man in der objektorientierten Programmierung die Übertragung der Definition von Eigenschaften und Verhalten (Intension) von einer Klasse auf eine andere. Vererbung dient vor allem der Wiederverwendung von Code und damit der Ökonomie in der Softwareentwicklung.

Wenn man das Prinzip von Generalisierung und Spezialisierung vor Augen hat, dann ist die Vererbung eigentlich nur noch ein Mechanismus, der Definitionen von einer Klasse auf eine andere überträgt. So wird jede benannte Instanzvariable, die in einer Generalisierung deklariert ist, nicht nur für Instanzen dieser Generalisierung (so sie denn welche hat) angelegt, sondern auch für die Instanzen all ihrer Spezialisierungen. Analog stehen Methoden, die in einer Generalisierung definiert werden, auch ihren Spezialisierungen zur Verfügung, und zwar beinahe so, als wären sie in den Spezialisierungen definiert.

**Vererbung bei
Generalisierung und
Spezialisierung**

Spezialisierung und Vererbung scheinen also Hand in Hand zu gehen. Doch ist dies nur solange der Fall, wie man von der Spezialisierung ausgeht und die Vererbung als ökonomisches Abfallprodukt erhält. In der Praxis lässt man sich doch leider häufig von vordergründigen Gewinnerwartungen leiten und folgt der (vermeintlichen) Ökonomie der Vererbung, ohne dabei auf die Prinzipien von Generalisierung und Spezialisierung einzugehen. Obiges Beispiel von Pinguinen und Vögeln hatte schon gezeigt, zu welchen Komplikationen eine unbedachte Spezialisierung führen kann; nachfolgendes soll zeigen, zu was eine Fixierung auf Ausnutzung der Vererbung führt.

Ein Klasse **Quadrat** sei etwa wie folgt definiert:

Klasse	Quadrat
benannte Instanzvariablen	laenge
Instanzmethoden	
415	flaeche
416	^ laenge * laenge



417 umfang
418 ^ laenge * 4

Nun möchte man eine zweite Klasse **Rechteck** definieren und dabei ausnutzen, dass man so eine ähnliche Klasse, nämlich **Quadrat**, schon hat. Aus **Quadrat** übernehmen lässt sich nämlich die Instanzvariable **laenge**.

(Das Beispiel wurde absichtlich einfach gewählt, auch wenn es dadurch wenig überzeugend wirkt; das Problem sollte aber trotzdem klarwerden.)

Vererbung zur Ausnutzung von Ähnlichkeiten

Klasse	Rechteck
beerbt von	Quadrat
benannte Instanzvariablen	breite
Instanzmethoden	
419 flaeche	
420	^ laenge * breite
421 umfang	
422	^ laenge + breite * 2

Was die Instanzvariablen angeht, so braucht **Rechteck** die Instanzvariable **laenge** nicht neu zu definieren, sondern muss lediglich **breite** hinzufügen.

Allerdings können die Methoden zur Berechnungen von Fläche und Umfang nicht mitgeerbt werden, obwohl Quadrate und Rechtecke die Eigenschaft, über solche Merkmale zu verfügen, teilen. Die entsprechenden Methoden müssen also in **Rechteck** neu definiert werden. Man nennt das **Überschreiben**, weil die neuen Methoden mit den alten genau dies tun. Die Möglichkeit des Überschreibens ist häufig Voraussetzung dafür, dass man Vererbung überhaupt sinnvoll einsetzen kann.

Überschreiben von Geerbtem

Wenn man nun glaubt, man hätte gleichzeitig mit der Vererbung auch eine Spezialisierungs- bzw. Generalisierungsbeziehung geschaffen, weit gefehlt: Die Menge der Quadrate enthält die Menge der Rechtecke nicht, was ja eine charakteristische Begleiterscheinung der Generalisierung gewesen wäre. Dass die Intension von **Rechteck** umfangreicher ist als die von **Quadrat** (sie enthält eine Instanzvariable mehr), ist eine Täuschung: Ein Quadrat hat, genau wie ein Rechteck, vier Seiten, nur ist die Bedingung für diese vier Seiten in Quadraten die, dass sie alle gleich lang sind, so dass man sich drei Instanzvariablen sparen kann; für Rechtecke sind nur jeweils zwei Seiten gleich lang, so dass man sich nur zwei Instanzvariablen spart. Die Intension für Quadrate ist aber trotzdem restriktiver als die für Rechtecke (sie enthält eine zusätzliche Bedingung), so dass der inverse Zusammenhang von Intension und Extension auch für Quadrate und Rechtecke gilt: je größer die Intension, desto kleiner die Extension (und umgekehrt).

Vererbung ohne Generalisierung/ Spezialisierung

Das Problem mit der Vererbung ist nun, dass sie auf die oberflächliche Wiederverwendung von Elementen einer Klassendefinition ausgerichtet

Oberflächlichkeit der Vererbung



ist. Sie lässt dabei insbesondere den Zusammenhang der Extensionen der beteiligten Klassen, der für Generalisierung/Spezialisierung wesentlich ist, außer acht. Diese Ignoranz hat aber weitreichende Konsequenzen, die wir in Kapitel 26 von Kurseinheit 3 noch kennenlernen werden.

Man hätte nun auch umgekehrt verfahren und dabei das Prinzip von Generalisierung und Spezialisierung hochhalten können, indem man **Quadrat** von **Rechteck** erben lässt (wenn man akzeptiert, dass die Generalisierung **Rechteck** eigene Instanzen hat). Der Nachteil dieses Entwurfs wäre jedoch, dass dann auch **Quadrat** zwei Instanzvariablen für Seitenlängen hätte, obwohl ja eine ausgereicht hätte. Auf der anderen Seite hätte man die Methoden für Fläche und Umfang nicht überschreiben müssen, denn wenn **laenge** und **breite** gleich sind, unterscheiden sich die beiden obigen Implementierungen von **flaeche** und **umfang** im Ergebnis nicht. Man muss nur sicherstellen, dass in Instanzen von **Quadrat** **laenge** und **breite** tatsächlich immer gleiche Werte haben.

Umkehrung der Vererbungsrichtung

Nun kann man aber auch auf die Idee kommen, die zu viel geerbte Instanzvariable **breite** einfach wieder zu löschen. Tatsächlich ist dies vom Standpunkt der Vererbung aus kein Problem: Genauso, wie man Teile der Definition überschreiben kann, kann man sie auch löschen. Im konkreten Fall der Klasse **Quadrat**, die von **Rechteck** erbaut wird, müsste man mit dem Löschen von **breite** aber auch die Methoden **flaeche** und **umfang** überschreiben. (Das Löschen von Methoden wäre auch möglich, wird hier aber nicht gebraucht.)

Löschen von Geerbtem

Was bleibt, ist ein Eindruck von Beliebigkeit bei der Vererbungsrichtung, die für Generalisierung/Spezialisierung nicht existiert. In gewisser Weise spiegeln Generalisierung/Spezialisierung und Vererbung auch zwei verschiedene Weltsichten wider: Generalisierung/Spezialisierung steht für die Ordnung eines Systems von Klassen mit Blick von *außen* und für das Ganze (die sog. *Client-Schnittstelle*), Vererbung für die Pragmatik des Programmierens mit Blick von *innen* und einem Fokus auf Wiederverwendung (die *Vererbungsschnittstelle*). Vererbung stellt eine Art genetischen Zusammenhang zwischen Klassen dar, der deren Entstehung aus Vorhandenem widerspiegelt, Generalisierung/Spezialisierung eher eine abstrakte Ordnung. Vererbung bringt Komplexität in ein System, Generalisierung/Spezialisierung versucht, sie durch Strukturierung zu reduzieren. Wie Sie gesehen haben, führen beide Sichten nicht automatisch zum selben Ergebnis; sie zu vereinen ist die hohe Kunst des objektorientierten Entwurfs.

Außen- und Innensicht

10.2 Vererbung in prototypenbasierten Sprachen

In der klassenbasierten Form der objektorientierten Programmierung ist die Vererbung an Klassen gebunden: Selbst wenn sich die Definitionen eigentlich auf die Instanzen der Klassen beziehen, so ist es doch die Klasse, die Teile ihrer Definition (Intension) von anderen erbt. Im Gegensatz dazu ist die Vererbung in prototypenbasierten objektorientierten Pro-



grammiersprachen, in denen es ja keine Klassen gibt, vollständig zwischen Objekten definiert: Jedes Objekt gibt eines oder mehrere andere an, deren Eigenschaften und Verhalten es übernimmt. Dabei kann es geerbte Teile der Definition überschreiben und auch löschen.

Auf den ersten Blick scheint es so, als sei dies sogar der natürlichere Weg der Vererbung: Schließlich findet in der Natur Vererbung ja auch ausschließlich zwischen Individuen statt, ja genaugenommen gibt es so etwas wie biologische Klassen (Arten etc.) in der Natur überhaupt nicht³⁶, denn es differenzieren sich ständig einzelne „Arten“ zu neuen und es ist nicht ausgeschlossen, dass einmal ausdifferenzierte Arten irgendwann wieder verschmelzen. Abgesehen davon ist, wie bereits in Kapitel 7 erwähnt, die reale Existenz von Allgemeinbegriffen strittig (der *Universalienstreit*).

Natürlichkeit der Vererbung zwischen Instanzen

Man kann dem freilich entgegenhalten, dass man als Programmiererin ja auch keine einzelnen Objekte, sondern Klassen entwirft, die damit die eigentliche „Schöpfung“ der objektorientierten Weltsicht abgeben. Auch sind objektorientierte Programme nicht für die Ewigkeit gemacht, sondern unterliegen der ständigen Anpassung, eben der Evolution, und somit sind auch Klassendefinitionen im ständigen Wandel. Eine Übertragung der Vererbung auf Klassen ist also nicht vollkommen unnatürlich.

Klassen als Geschöpfe der Programmierung

Nicht zuletzt muss man auch erkennen, dass viele Anwendungsdomänen, für die programmiert wird, aus massenhaft gleichen Objekten bestehen, die durch den klassenbasierten Ansatz besser abgedeckt werden als durch den prototypenbasierten (vgl. die entsprechenden Kommentare zur Klassifikation in Abschnitt 7.1). Und so macht denn auch die Vererbung unter Instanzen das Nachvollziehen (und Debuggen) eines Programms eher noch schwieriger als die Vererbung unter Klassen ohnehin schon (s. Kapitel 56 in Kurseinheit 6).

Chaos durch individuenbasierte Vererbung

10.3 Abstrakte Klassen

Die Genera Aristoteles' sind allesamt abstrakt — es gibt keine Säugetiere, die nicht Mensch oder Hund oder Katze oder was Konkretes auch immer wären. Übertragen auf die objektorientierte Programmierung hieße das: Generalisierungen, also Klassen, die aus Generalisierungen hervorgegangen sind, haben selbst keine Instanzen, sind also insbesondere nicht *instanziierbar*.

In der objektorientierten Programmierung nennt man nicht instanzierbare Klassen **abstrakt**. Der Grund für die mangelnde Instanzierbarkeit

abstrakte und konkrete Klassen

³⁶ So ist zwar die Definition einer Art als diejenige biologische Kategorie, deren Exemplare (Individuen) untereinander fortpflanzungsfähige Nachkommen zeugen können, sinnvoll, doch unterliegen derart angelegte Artendefinitionen dem erdgeschichtlichen Wandel, wie sich schon daraus ableiten lässt, dass alles Leben aus den ersten Einzellern entstanden ist.



ist jedoch häufig kein konzeptueller (wie beispielsweise, dass es sich bei einer Klasse um eine Generalisierung handelt und sie daher nicht instanzierbar sein sollte), sondern ein rein technischer: Abstrakten Klassen fehlen in der Regel Angaben, die das Verhalten ihrer Instanzen vollständig spezifiziert und diese somit brauchbar machen würden, so dass Instanzen dieser Klassen, wenn es sie denn geben würde, unvollständig definiert wären und zu Laufzeitfehlern führen würden. Diese fehlenden Eigenschaften werden erst in den Klassen geliefert, die von den abstrakten erben (s. nächstes Kapitel), wobei die Idee ist, dass sich die Eigenschaften von Klasse zu Klasse unterscheiden. Klassen, die nicht abstrakt sind, die also eigene Instanzen haben können, nennt man **konkret**.

Ein typisches Beispiel für eine abstrakte Klasse in SMALLTALK ist die Klasse **Collection**. Sie ist (in Auszügen) wie folgt definiert:

Beispiel einer abstrakten Klasse: **Collection**

Klasse	Collection
benannte Instanzvariablen	
indizierte Instanzvariablen	nein
Instanzmethoden	
423	add: anObject
424	"Answer anObject. Add anObject
425	to the receiver collection."
426	^ self implementedBySubclass
427	addAll: aCollection
428	"Answer aCollection. Add each element of
429	aCollection to the elements of the receiver."
430	aCollection do: [:element self add: element].
431	^ aCollection

Man erkennt schon am Fehlen von Instanzvariablen, dass es mit der Implementierung von **Collection** nicht weit her sein kann — Instanzen wären schlicht zustandslos, weswegen sie kaum zu gebrauchen wären. Besonders deutlich wird die Abstraktheit jedoch an der Implementation der Methode **add::**: Hier wird, anstatt etwas Entsprechendes zu tun, die Methode **implementedBySubclass**³⁷ aufgerufen, die eine Fehlermeldung ausgibt. Jede, die mit einer direkten Instanz von **Collection** arbeiten würde und die Methode **add:** (oder **addAll::**; auf die Bedeutung von **self** im Kontext von abstrakten Klassen und Vererbung gehen wir gleich und später dann in Abschnitt 12.1, „Nachrichten an **self**“, noch einmal ein) darauf aufrufen wollte, würde enttäuscht.

Allerdings erst zur Laufzeit. Viele andere Sprachen verlangen daher, dass man abstrakte Klassen mit einem Schlüsselwort, z. B. **abstract**, markiert, und verbieten dann (per Compiler), die Klasse zu instanziieren. Das geht in SMALLTALK

Verbot der Instanziierung

³⁷ In anderen SMALLTALK-Dialektken heißt die Nachricht **subclassResponsibility**.



jedoch nicht, da Klassen auch Objekte sind und daher in Variablen gespeichert werden können, denen man dann einfach `new` schicken kann, ohne dass der Compiler wissen könnte, welches Objekt die Variable nun gerade bezeichnet.

Selbsttestaufgabe 10.1

Probieren Sie es gleich aus, d. h., weisen Sie eine Klasse einer Variable zu und schicken sie dem durch die Variable bezeichnetem Objekt die Methode `new`!

Nun erfolgt der Hinweis, dass man eine abstrakte Klasse instanziert hat, in SMALLTALK nicht nur erst zur Laufzeit, sondern auch da erst zum spätestmöglichen Zeitpunkt, nämlich wenn man eine nicht implementierte Methode aufzurufen versucht. Was man tun könnte, um zu verhindern, dass Instanzen einer abstrakten Klasse überhaupt erzeugt werden, ist, die Konstruktoren, insbesondere `new` und `new:`, entsprechend zu überschreiben (vgl. Abschnitt 8.2).³⁸ Allerdings verhindert man damit zunächst auch die Instanziierung der Klassen, die von `Collection` erben, die natürlich nicht alle abstrakt sein sollen. Diese müssten dann `new` und `new:` wieder neu einführen, was aber kaum zumutbar ist, zumal `new` und `new:` primitive Methoden (s. Abschnitt 4.3.7) aufrufen. (Vgl. hierzu auch die Grenzen der Verwendung von `super` in Abschnitt 12.2.)

Verhinderung der Instanziierung

Man könnte in SMALLTALK die Methode `add:` in der Klasse `Collection` natürlich auch ganz weglassen.³⁹ Ein Aufruf von `add:` auf einer Instanz von `Collection` oder einer ihrer Subklassen würde dann zum Aufruf von `doesNotUnderstand` und der Ausgabe einer entsprechenden Fehlermeldung führen. Allerdings wäre diese Fehlermeldung für die Programmiererin weniger aufschlussreich: Sie wüsste nicht, ob sie einfach nur einen falschen Methodennamen verwendet hat (ihr Fehler) oder ob die Programmiererin einer Subklasse von `Collection` vergessen hat, die Methode `add:` zu implementieren (jemand anderes Fehler). Eine Methode wie `add:` in `Collection` vorzusehen, die auf ein Versäumnis hinweist, so es denn eines gibt (es könnte ja auch sein, dass man versehentlich eine Instanz von `Collection` erzeugt hat und darauf `add:` ausführt, obwohl `add:` für alle erbenden Klassen implementiert ist — das erlaubt dann die Fehlermeldung nicht zu unterscheiden) ist schon sinnvoll. Die Laufzeitfehlermeldungen von SMALLTALK ersetzen also gewissermaßen die Compiler-Fehlermeldungen anderer Sprachen. Die entsprechenden Grundlagen werden Ihnen in Kurseinheit 3 begegnen.

abstrakte Klassen faktorisieren gemeinsame Implementierungen heraus

³⁸ `new` und `new:` werden geerbt — wie und von wo, das ist Gegenstand von Abschnitt 11.4.

³⁹ Dies ist möglich, weil der Compiler bei Methodenaufrufen überhaupt nicht prüft, ob die Methode auch vorhanden ist, selbst wenn sie auf `self` aufgerufen wird (und damit schon klar ist, welche Klassen die Methode eigentlich haben müssten). In anderen Sprachen ist das anders.



Man mag sich fragen, warum es eine *abstrakte Klasse* wie `Collection` überhaupt gibt, wenn sie doch keine Instanzen haben soll.⁴⁰ Bereits das obige Beispiel gibt eine erste Antwort: Weil es auch in abstrakten Klassen Methoden gibt, die voll ausimplementiert sind (z. B. `addAll:`) und die dann in den Subklassen, auf die sie vererbt werden, nicht wiederholt werden müssen. Tatsächlich ist alles, was eine erbende Klasse tun muss, um in den Genuss einer funktionierenden Methode `addAll:` zu kommen, eine Implementierung von `add:` zu liefern. Die abstrakte Klasse faktorisiert also die Gemeinsamkeiten mehrerer Klassen heraus und markiert gleichzeitig, was ihre erbenden Klassen noch nachtragen müssen: Alle Methoden, die von anderen Methoden der Klasse aufgerufen werden (wie eben `add:` von `addAll:`), die aber (z. B. mangels Instanzvariablen wie im Fall von `Collection`) in der Klasse noch nicht implementiert werden können, deren Aufruf auf Instanzen der abstrakten Klassen somit einen entsprechenden Fehler liefern würde.

Der Aufruf einer abstrakten, d. h. in der Klasse nicht implementierten Methode aus der Klasse selbst heraus wie in Zeile 430 (mit `self` als Empfänger) ist ein gängiges Muster der objektorientierten Programmierung. Man nennt es auch **offene Rekursion** (engl. open recursion), da der Aufruf auf dem Objekt selbst erfolgt (also gewissermaßen rekursiv ist), aber an der aufrufenden Stelle noch nicht klar (offen) ist, welche (erbende) Klasse die Implementierung liefert. Dieses Muster, auf das wir in Abschnitt 12.1 im Rahmen des dynamischen Bindens noch einmal zurückkommen werden, lässt sich auch einsetzen, um das oben beschriebene Dilemma von `Quadrat` und `Rechteck` aufzulösen:

offene Rekursion

Klasse	<code>Rechteck</code>
benannte Instanzvariablen	
Instanzmethoden	<pre> 432 laenge 433 ^ self implementedBySubclass 434 breite 435 ^ self implementedBySubclass 436 flaeche 437 ^ self laenge * self breite 438 umfang 439 ^ self laenge + self breite * 2 </pre>

Die Definitionen von `Quadrat` und `Rechteck` fallen dann knapp aus und kommen ohne inhaltliche Veränderungen daher (lediglich die noch nicht implementierten Methoden müssen nachgeliefert werden):

Klasse	<code>Quadrat</code>
--------	----------------------

⁴⁰ Konzeptuelle Gründe wie etwa die Existenz einer sinnvollen Generalisierung lassen wir hier einmal außen vor.



beerbt Klassen

Rechteck

benannte Instanzvariablen

laenge

Instanzmethoden

440 **laenge**

441 ^ laenge

442 **breite**

443 ^ laenge

Klasse

NichtQuadratischesRechteck

beerbt Klassen

Rechteck

benannte Instanzvariablen

laenge breite

Instanzmethoden

444 **laenge**

445 ^ laenge

446 **breite**

447 ^ breite

Es sollte klar sein, dass alle Methoden nur für Quadrate und Rechtecke definiert sind.

Man beachte, dass dieses Beispiel auch die Kriterien von Generalisierung und Spezialisierung erfüllt: Die Menge der Quadrate und die der nicht quadratischen Rechtecke ist in der Menge der Rechtecke enthalten und die Definition der beiden stellt jeweils eine Erweiterung letzterer dar. Ein Überschreiben oder sogar Löschen wird nicht nötig (sieht man mal vom Überschreiben der „nicht implementierten“ Methoden ab).

11 Superklassen und Subklassen

Wenn Sie sich schon vor diesem Kurs mit der objektorientierten Programmierung befasst haben, dann fragen Sie sich vielleicht, warum die Begriffe der Super- und Subklasse bislang nicht fielen. Das liegt daran, dass diese in verschiedenen Programmiersprachen verschiedene Bedeutungen haben, während die Begriffe der Generalisierung und Spezialisierung sowie die der Vererbung und der abstrakten Klassen recht einheitlich interpretiert werden.

Die **Subklassenbeziehung** ist, genau wie Generalisierung, Spezialisierung und Vererbung, eine Beziehung zwischen Klassen. Die beiden Enden

Eigenschaften der
Subklassenbeziehung

der Beziehung vergeben die Rollen **Superklasse** bzw. **Subklasse**; die Präfixe legen nahe, dass die Subklassenbeziehung eine **Klassenhierarchie** aufbaut, in der die Superklassen über den Subklassen stehen. Außerdem ist die Subklassenbeziehung transitiv: Wenn A eine Subklasse von B ist und B eine von C, dann ist A auch eine Subklasse von C. Analoges gilt natürlich auch für Superklassen. Man spricht übrigens von einer **direkten Subklasse** bzw.



von einer **direkten Superklasse**, wenn es keine weitere Klasse gibt, die in der Subklassenbeziehung dazwischen steht. Die Subklassenbeziehung ist (anders als die Subtypenbeziehung; vgl. Kapitel 26) nicht reflexiv (irreflexiv) — eine Klasse kann also keine Subklasse von sich selbst sein.

11.1 Bedeutung der Subklassenbeziehung

Die Bedeutung der Subklassenbeziehung variiert von Sprache zu Sprache. Wie Sie sich vielleicht schon gedacht haben, kann man die Subklassenbeziehung mit der Spezialisierungsbeziehung gleichsetzen oder auch mit der Vererbung; es sind aber auch noch andere Definitionen möglich. Tatsächlich wird die hier als Subklassenbeziehung eingeführte Beziehung zwischen Klassen auch gar nicht immer so genannt; entsprechend heißen dann die Rollen auch nicht Sub- und Superklasse, sondern z. B. **abgeleitete Klassen** und **Basisklassen**. Im Englischen sind hierfür neben *Derived class* und *Base class* auch die Begriffe *Child class* bzw. *Parent class* in Gebrauch.

In SMALLTALK wird die Subklassenbeziehung mit der Vererbungsbeziehung gleichgesetzt. Eine Subklasse erbt demnach alle Instanzvariablen und Methoden ihrer Superklasse. Dass sie darüber hinaus auch noch ihre Klassenvariablen und -methoden erbt, ist nicht selbstverständlich; dies wird in Abschnitt 11.4 noch genauer beleuchtet. Wichtig ist hier, festzuhalten, dass durch eine existierende Subklassenbeziehung zwischen zwei Klassen nicht ausgedrückt wird, dass die Subklasse eine Spezialisierung der Superklasse ist oder gar die Superklasse eine Generalisierung der Subklasse. Dies sicherzustellen obliegt der Verantwortung der Programmiererin.

**Subklassenbeziehung
in SMALLTALK**

Jede neue Klasse, die in einem SMALLTALK-System angelegt wird, muss direkte Subklasse genau einer Klasse sein — es ist deshalb notwendig, dass beim Erzeugen einer neuen Klasse die Superklasse mit angegeben wird. Da wie bereits mehrfach erwähnt die SMALLTALK-Programmierung nicht dateibasiert ist, sondern mittels eines dafür vorgesehenen Browsers erfolgt, gibt es zum Zweck der Angabe der Superklasse auch kein spezielles Schlüsselwort wie beispielsweise **extends**, das die Subklassenbeziehung ausdrückt: Man legt vielmehr eine neue Klasse an, indem man ihrer Superklasse eine entsprechende Nachricht schickt. Eine dazugehörige Methode hatten Sie bereits in Abschnitt 8.4 gesehen.

**Erzeugung von
Klassen als
Subklassen**

Damit eine Subklassenbeziehung zwischen zwei Klassen zulässig ist, müssen deren Definitionen bestimmte Bedingungen einhalten. In SMALLTALK gilt dabei für neue, benannte Instanz- und Klassenvariablen, dass sie nicht dieselben Namen haben dürfen wie Variablen, die bereits in (direkten oder indirekten) Superklassen deklariert wurden. Für indizierte Instanzvariablen gilt, dass wenn die Superklasse solche hat, sie auch die Subklasse haben muss. Methodendefinitionen hingegen, die dieselbe *Methodensignatur* verwenden, überschreiben einfach die geerbten Methoden. Entsprechende Regeln sind in anderen Programmiersprachen zum Teil erheblich komplexer.

**Bedingungen für die
Zulässigkeit einer
Subklassenbeziehung**



Da die Subklassenbeziehung auch in SMALLTALK nicht reflexiv ist, muss es mindestens eine Klasse geben, die keine Subklasse ist (und entsprechend keine Superklasse hat). Es ist die Klasse **Object**, die oberste aller Superklassen. In ihr sind die Definitionen angelegt, die den Instanzen aller Klassen zugutekommen sollen (also z. B. die Methode **printString**). Diese Methoden werden per Vererbung auf alle anderen Klassen übertragen, wodurch sie deren Instanzen zur Verfügung stehen. Eine ganze Reihe nützlicher Methoden, die in **Object** definiert sind, werden wir in Kapitel 14 kennenlernen.

11.2 Mechanismus der Vererbung von Superklassen auf Subklassen

Es stellt sich die Frage, wie der Mechanismus der Vererbung genau umgesetzt wird. Eine Möglichkeit wäre z. B., die Definition einer Superklasse per Kopieren und Einfügen auf ihre Subklassen zu übertragen. Das wäre zwar möglich und würde auch die Semantik der Vererbung korrekt wiedergeben, würde aber das (technische) Problem mit sich bringen, dass bei einer Änderung einer Superklasse auch alle ihre Subklassen mit geändert werden müssten.

Eine weitere Möglichkeit wäre, für jede Instanz einer Subklasse automatisch je eine Instanz aller ihrer Superklassen mit zu erzeugen und diese Instanzen zu einer zu vereinen. Diese Umsetzung der Vererbung steht jedoch mit dem Konzept der Identität von Objekten in Konflikt: Ein Objekt einer Subklasse hätte auf einmal mehrere Identitäten, und zwar eine für sich selbst und eine pro Superklasse, von der sie erbt. Auch das wäre problematisch.

Stattdessen wird die Vererbung in SMALLTALK und vielen anderen objekt-orientierten Programmiersprachen als ein Aufteilen der Klassendefinitionen realisiert: Vereinbarungen, die in einer Klasse getroffen wurden, gelten automatisch auch für alle Subklassen, es sei denn, diese spezifizieren etwas anderes. Dabei werden die Vereinbarungen nicht übertragen (wie per Kopieren und Einfügen), sondern einfach nur mitbenutzt.

**Vererbung als Teilen
der
Klassendefinitionen**

11.3 Löschen von Methoden

Wie bereits in Abschnitt 10.3 erwähnt, wird die Programmiererin, die eine abstrakte, weil unvollständige, Klasse instanziert, irgendwann damit bestraft, dass das Versenden einer Nachricht an die entsprechende Instanz zu einer Fehlermeldung führt, die ihr (per **subclassResponsibility** oder **implementedBySubclass**, die, genau wie **doesNotUnderstand:**, in der Klasse **Object** definiert ist) anzeigt, dass die Methode (erst) in einer Subklasse implementiert werden sollte. Dummerweise bekommt die Programmiererin diesen Hinweis erst zur Laufzeit des Programms zu Gesicht, also dann, wenn es schon zu spät ist (es sei denn, man testet gerade). Man erkennt hieran sehr schön den interaktiven Geist des SMALLTALK-Systems, insbesondere die Philosophie, nach der Programmieren nichts



weiter ist als das iterative Zurechbiegen und Erweitern eines bereits bestehenden, funktionierenden Systems. Man muss eine Weile damit gespielt haben, um diesem Charme zu erliegen.

Selbsttestaufgabe 11.1 (für JAVA-Fans)

Überschreiben Sie die Methode `doesNotUnderstand`: so, dass man beim Versenden einer Nachricht an `nil` eine Meldung „Null pointer exception“ erhält. Achtung: Speichern Sie vorher unbedingt Ihr Image und stellen Sie es nach der Bearbeitung der Aufgabe wieder her!

Wenn man sich erst einmal damit abgefunden hat, dass man als Programmiererin Methoden schreibt, die ausschließlich dem Zweck dienen, sich selbst oder eine Kollegin auf Programmierfehler hinzuweisen, dann erscheint einem eine weitere SMALLTALK-Konvention geradezu als elegant, nämlich die, geerbte Methoden durch überschreiben auszulöschen. Tatsächlich ist genau hierfür eine weitere Methode in der Klasse `Object` mit Namen „`shouldNotImplement`“ vorgesehen, die zu einer entsprechenden Fehlermeldung führt. Eine Klasse, die also eine geerbte Methode löschen möchte, überschreibt diese einfach mit

**Löschen durch
Überschreiben**

448 `self shouldNotImplement`

im Rumpf. Bevor Sie jetzt als disziplinierte Programmiererin den Stab über SMALLTALK brechen, erlauben Sie noch den Hinweis, dass der Wunsch, geerbte Methoden zu löschen, direkte Folge der vorrangigen Orientierung an Vererbung ist, die bereits oben kritisiert wurde: Wäre die Superklasse auf Grundlage des Prinzips der Generalisierung ausgewählt worden, käme man gar nicht in die Verlegenheit, Methoden löschen zu wollen, denn alles, was für die Generalisierung sinnvoll ist, ist grundsätzlich auch für ihre Spezialisierungen sinnvoll, oder die Generalisierung ist keine Generalisierung. Außerdem haben Sie auch in Sprachen mit starker Typprüfung, in denen das Löschen von Methoden nicht möglich ist, als Programmiererin immer die Freiheit, eine Methode so zu überschreiben, dass sie garantiert nichts tut, was mit der Idee der Klasse, von der sie ererbt ist, in Einklang zu bringen wäre. Auch hier wären Laufzeitfehler die unvermeidbare Folge. Mehr dazu im Kurseinheit 3; hier sei nur soviel bemerkt wie, dass wenn man sich bei der Organisation seiner Klassenhierarchie auf das Prinzip der Generalisierung stützt, dass man dann auch nicht in die Verlegenheit kommt, Methoden löschen zu wollen.

11.4 Subklassenhierarchie und Vererbung unter Metaklassen

Vererbung ist nicht auf die Klassen der Ebene 1 beschränkt — es können in SMALLTALK vielmehr auch Metaklassen, die ja ebenfalls Klassen sind (s. Kapitel 8), voneinander erben. Da Metaklassen aber bei der Erzeugung von Klassen automatisch angelegt werden (und auch keine eigenen Namen haben), hat die Programmiererin auch keinen direkten Einfluss auf die Vererbungshierarchie der Metaklassen. Vielmehr wird diese automatisch parallel zur Vererbungshierarchie der Klassen, die Instanzen der Metaklassen sind, angelegt. Dies hat zur



Folge, dass in SMALLTALK neben den Instanzvariablen und -methoden auch die Klassenvariablen und -methoden von einer Klasse auf ihre Subklassen vererbt werden.

Da in SMALLTALK jede Klasse direkte oder indirekte Subklasse von **Object** ist und die Subklassenhierarchie der Metaklassen parallel zu der ihrer Klassen angelegt ist, erbt jede Metaklasse in SMALLTALK automatisch von **Object class**, der Metaklasse von **Object**. Was läge also näher, als die Klassenmethoden, die allen Klassen zur Verfügung stehen sollen — darunter auch die beiden Standardkonstruktoren **new** und **new:** — in **Object** (genauer: als Instanzmethoden von **Object class**) zu definieren?

**Ursprung von
Klassenmethoden
aller Klassen**

Nun gibt es ja schon, wie bereits in Fußnote 29 oben erwähnt, in SMALLTALK zwei Arten von Objekten, nämlich solche, die instanziierbar sind (also Klassen) und solche, die es nicht sind. Darüber hinaus gibt es auch noch eine Unterscheidung zwischen Klassen, die Metaklassen sind, und solchen, die es nicht sind — bei allen Gemeinsamkeiten von Klassen und Metaklassen muss man z. B. von Klassen neue Subklassen bilden können, von Metaklassen jedoch nicht. Diese Unterscheidungen müssen schließlich irgendwo getroffen werden. Und so kommt es, dass **Object class** nicht die Wurzel der Vererbungshierarchie der Metaklassen ist (kann sie sowieso nicht, denn auch sie muss eine Subklasse von **Object** sein!), sondern selbst von einer für diesen Zweck vorgesehenen Klasse erbt. Aus demselben Grund, aus dem die Klasse **Object** „Object“ und die Klasse **Metaclass** „Metaclass“ heißt, heißt diese Klasse „Class“: Es gilt nämlich für jede Instanz dieser Klasse, dass sie eine Klasse ist. Man beachte übrigens, dass **Class**, auch wenn sie die Superklasse aller Metaklassen ist, selbst keine Metaklasse ist, denn sonst müsste **Class** ja als Superklasse von **Object class** und wegen der parallelen Vererbungshierarchie von Metaklassen und Klassen die (Meta-)Klasse einer Klasse sein, die Superklasse von **Object** ist. Ist sie aber nicht. Außerdem ist, wie man sich leicht überzeugen kann, die Klasse von **Class** die Klasse **Class class** und erst **Class class** eine Metaklasse. Zugegebenermaßen etwas kompliziert.

Selbsttestaufgabe 11.2

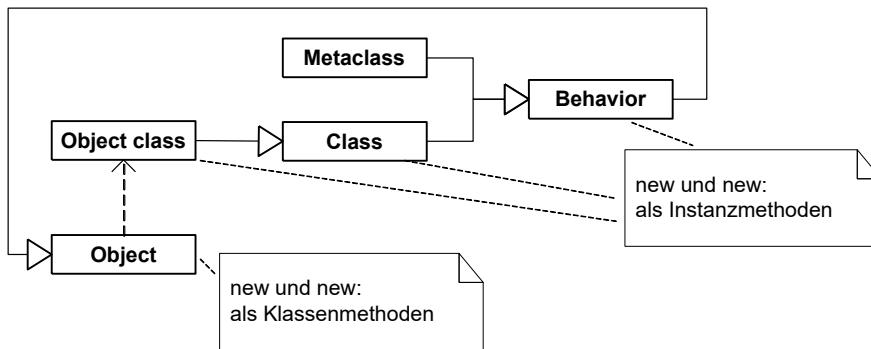
Finden Sie für das SMALLTALK-System Ihrer Wahl heraus, wie die Zusammenhänge der Klassen **Object**, **Class** und **Metaclass** sowie derer jeweiligen Metaklassen **Object class**, **Class class** und **Metaclass class** sind. Benutzen Sie dazu die Methoden **allSuperclasses**, **allSubclasses** und **isKindOf:**. (Um zu testen, ob ein Objekt in einer Aufzählung, wie sie von **allSuperclasses** und **allSubclasses** zurückgeliefert wird, enthalten ist, können Sie die Methode **includes:** verwenden.)

Die Klasse **Class** steht in der Vererbungshierarchie SMALLTALKS neben der Klasse **Metaclass**. Gemeinsam erben sie von der Klasse **Behavior** (in SMALLTALK-80 und direkten Derivaten indirekt, über die Klasse **ClassDescription**), in der endlich, neben vielen anderen Methoden, **new** und **new:** definiert sind. Man beachte, dass diese Methoden als Instanzmethoden deklariert sind; da sie aber in der Vererbungshierarchie SMALLTALKS von den Metaklassen der Klassen geerbt werden (z. B. **Object class**),

**Einordnung in die
Klassenhierarchie**

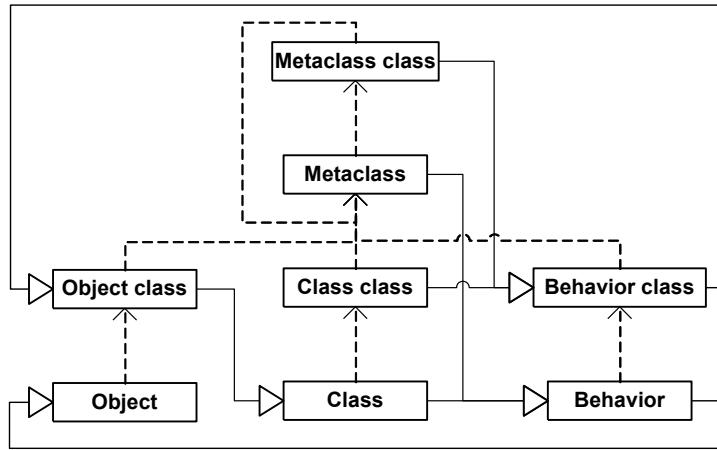


stehen sie in den Klassen als Klassenmethoden zur Verfügung. `new` und `new:` werden also in der Praxis immer an Klassen geschickt.



Nun ist es in SMALLTALK so, dass auch alle Metaklassen (Ebene 2) und die Klasse **Metaclass** (Ebene 3) Subklassen von **Object** sein müssen. Dabei kommt es natürlich zu einem fröhlichen Ebenenmix, der nur sehr schwer nachzuvollziehen ist. Wenn Ihnen das Probleme bereitet, brauchen Sie sich keine Sorgen zu machen, denn Sie haben einen guten Grund: Legt man nämlich wie schon in den Abschnitten 7.3 und 9.1 eine mengentheoretische Interpretation von Klassen als Mengen ihrer Instanzen und von Superklassen als Obermengen der Mengen, für die ihre Subklassen stehen, zugrunde, dann ergibt sich, da **Object class**, die Metaklasse von **Object**, auch eine Subklasse von **Object** sein muss, dass die Menge von **Object** in sich selbst enthalten sein müsste, was aber aus theoretischen Gründen nicht möglich ist. Schon daran erkennt man, dass beim Entwurf von SMALLTALK der pragmatische Gesichtspunkt der Vererbung im Vordergrund stand und nicht etwa der konzeptuelle der Generalisierung.

Konflikt mit mengentheoretischer Interpretation



11.5 Dominanz der Vererbung

SMALLTALK stammt aus einer Zeit, in der man mit der objektorientierten Programmierung noch relativ wenig praktische Erfahrung gesammelt hatte. Damals war man der Ansicht, einer der Hauptvorteile der objektorientierten Programmierung sei die Wiederverwendung von Code durch Vererbung. Nun hat die Vererbung, wie in Abschnitt 10.1 dargelegt, durch-



aus etwas mit der auf Generalisierung bzw. Spezialisierung beruhenden Abstraktionshierarchie zu tun: Der allgemeinere Begriff (die Superklasse) überträgt (vererbt) alle seine Eigenschaften auf die spezielleren Begriffe (Subklassen), der speziellere erbt sie von den allgemeineren. Dies liegt in der Natur der Sache. Problematisch wird es jedoch, sobald man den kausalen Zusammenhang umkehren und von einer möglichen Vererbung auf eine Generalisierung/Spezialisierung schließen will: Nur weil eine Klasse (zufällig) Eigenschaften einer anderen gebrauchen könnte, heißt das noch lange nicht, dass die erbende Klasse auch eine Spezialisierung der vererbenden ist. Ein klassisches Beispiel hierzu hatten wir mit der Ableitung der Klasse **Rechteck** von der Klasse **Quadrat** bereits kennengelernt; den unangenehmen Folgen solch vererbungsorientierter Vorgehensweisen werden wir in der nächsten Kurseinheit noch begegnen.

12 Dynamisches Binden

Wie bereits in den Abschnitten 4.3.2 und 4.5.2 in Kurseinheit 1 angerissen, verbirgt sich hinter dem Nachrichtenversand ein dynamisch gebundener Methodenaufruf. Dabei ist die Auswahl der Methode nicht nur vom Nachrichtenselektor, sondern auch vom Empfängerobjekt abhängig. In Abschnitt 11.2 hatten wir bereits angedeutet, wie in Superklassen definierte Methoden für ihre Subklassen zugreifbar sind; hier schauen wir uns nun etwas genauer an, wie die dynamische Bindung von Methodenaufrufen vorstatten geht.

Wenn eine Methode auf einem Empfängerobjekt aufgerufen wird, wird zunächst geprüft, ob die Methode im zur Klasse des Empfängers gehörenden Methodenwörterbuch enthalten ist. Dies kann man auch selbst tun: Es gibt dafür in der Klasse **Behavior** eine Instanzmethode

Ablauf eines dynamisch gebundenen Methodenaufrufs

```
449 includesSelector: aSymbol  
450   ^ self methodDictionary includesKey: aSymbol
```

oder so ähnlich (je nach System), die somit allen Klassen (als Klassenmethode) zur Verfügung steht. (**Behavior** ist ja eine Superklasse von **Class**, die wiederum Superklasse aller Metaklassen ist, einer derer jede Klasse eine Instanz ist, so dass alle Klassen die Methode **includesSelector:** verstehen.)

Wird die Methode gefunden, dann wird sie ausgeführt. Wird sie nicht gefunden, wird zunächst in der direkten Superklasse der Klasse des Objekts weitergesucht und dann in deren direkter Superklasse usw. bis zur Klasse **Object**. Sobald die Methode gefunden wird, wird sie ausgeführt. Wird die Methode auch in **Object** nicht gefunden, kommt es zum bereits (in den Abschnitten 4.3.2 und 11.3) erwähnten Versenden der Nachricht **doesNotUnderstand:** an den ursprünglichen Empfänger mit der ursprünglichen, problematischen Nachricht als Argument.



Man beachte, dass selbst wenn auf den Empfang einer Nachricht die Methode einer Superklasse des Empfängerobjekts ausgeführt wird, das Objekt, auf dem sie ausgeführt wird, das Empfängerobjekt bleibt. Da die Methode jedoch in einer Superklasse kompiliert wurde (und zum Zeitpunkt der Kompilierung die Subklassen u. U. noch gar nicht existierten), kann die Methode nur auf die Instanzvariablen zugreifen, die für die Objekte der entsprechenden Klasse zugreifbar sind. Instanzvariablen, die erst in der Klasse des Objekts hinzugekommen sind, sind für die Methode also nicht (direkt) sichtbar. Gleichwohl — und das wird häufig nicht verstanden — handelt es sich immer noch um das ursprüngliche Empfängerobjekt, das auch immer noch Instanz seiner Klasse ist. Die gerade ausgeführte Methode betrachtet es lediglich wie ein Objekt der Klasse, in der sie (die Methode) definiert ist. Dies hat auch Auswirkungen auf die Bedeutung der Pseudovariable **super**, wie wir noch sehen werden.

Der Suchalgorithmus ist, genau wie der Methodenaufruf selbst, aus Effizienzgründen direkt in der virtuellen Maschine implementiert. Die Implementierung ist jedoch im wesentlichen äquivalent zu der der Methode **canUnderstand:**, die genau wie **includesSelector:** in der Klasse **Behavior** definiert ist:

```
451 canUnderstand: aSelector
452   (self includesSelector: selector) ifTrue: [^true].
453   superclass == nil ifTrue: [^false].
454   ^superclass canUnderstand: selector
```

Man beachte übrigens, wie wenig Aufwand es ist, aus dem klassenbasierten Methoden-Lookup einen objektbasierten zu machen: Man muss dazu lediglich jedem einzelnen Objekt sein eigenes Methodenwörterbuch zur Verfügung stellen. Wenn man zusätzlich noch Objekte von Objekten anstatt Klassen von Klassen erben lässt, dann hat man schon die prototypenbasierte Form der objektorientierten Programmierung. Der Unterschied ist also technisch nicht besonders groß — konzeptuell hingegen schon, denn mit den Klassen entfielen auch die sonst so nützlichen Begriffe von Generalisierung und Spezialisierung (von der Generalisierung von Objekten zu sprechen erscheint wenig sinnvoll).

**Unterschied zu
prototypenbasierter
Form der
objektorientierten
Programmierung**

Selbsttestaufgabe 12.1

Schreiben Sie eine Klasse **PrototypicalObject** und ändern Sie darin die Methoden **perform**, **perform:** etc. so ab, dass zunächst in einem jedem Objekt eigenen Methodenwörterbuch nachgeschlagen wird, ob es eine passende Methode für das Empfängerobjekt gibt. Was fehlt noch, damit Ihr SMALLTALK zu einem echten Hybriden (klassenbasierte plus prototypenbasierte Form der Objektorientierung) wird?

Eines der immer wieder vorgetragenen Hauptargumente gegen den Einsatz von SMALLTALK in der kommerziellen Programmierung ist der Umstand, dass das dynamische Binden wirklich vollkommen dynamisch ist: Dass einem Objekt eine Nachricht geschickt wird, die es nicht versteht, tritt immer erst zur Laufzeit zutage (s.

**Kritik an SMALLTALKS
dynamischer Prüfung**



Abschnitt 10.3 und 11.3).⁴¹ In den statisch typgeprüften Sprachen, die wir in den nächsten Kurseinheiten kennenlernen werden, ist das charakteristischerweise nicht so. Dem kann man entgegenhalten, dass die heutigen (auch) statisch typgeprüften Programmiersprachen wie JAVA, C# oder C++ sämtliche nicht ohne dynamische *Typumwandlungen* auskommen, die ebenfalls zu Laufzeitfehlern führen können. Tatsächlich ist es sowohl in SMALLTALK als auch in JAVA und C# (in C++ nur mit Einschränkungen; s. Abschnitt 51.5) nicht nur möglich, sondern sogar geboten, Laufzeitfehler da, wo möglich, zu vermeiden, indem man vor einem Methodenaufruf explizit prüft, ob ein Objekt die gewünschte Methode auch hat — in SMALLTALK mittels `canUnderstand:`, in JAVA et al. mittels eines entsprechenden Typtests vor einem *Down cast*. Die größere Flexibilität, die die objektorientierte Programmierung durch das dynamische Binden bietet, hat eben den Preis, dass bestimmte Laufzeitprüfungen durchgeführt werden müssen. Statische Typprüfung kann das Risiko von Typfehlern verringern, aber nicht ausschließen — gleichzeitig schränkt es die Flexibilität beim Programmieren ein, ein Umstand, der so manchen, der schon einmal größere Programme in SMALLTALK geschrieben hat, an der Verwendung typegeprüfter Sprachen stört.

12.1 Nachrichten an `self`

In SMALLTALK muss das Empfängerobjekt eines Nachrichtenversands immer explizit gemacht werden, selbst wenn sich die dazu passende Methode in derselben Klasse befindet. So kann also insbesondere `self` nicht (wie beispielsweise `this` in JAVA) weggelassen werden, wenn ein Objekt eine Nachricht an sich selbst schicken möchte. Wie bereits in Abschnitt 4.3.1 erwähnt, bezeichnet die Pseudovariable `self` immer den Empfänger der Nachricht, also dasjenige Objekt, auf dem die Methode, in deren Definition die Variable `self` vorkommt, gerade ausgeführt wird, und dessen Instanzvariablen zugreifbar sind. (Die einzige Ausnahme hiervon bilden Blöcke, in denen `self` sich auf den Empfänger des *Home context* bezieht; s. Abschnitt 4.4.1 in Kurseinheit 1).

Dabei ist allerdings zu beachten, dass die Klasse des durch `self` bezeichneten Objekts nicht unbedingt dieselbe sein muss, in der die gerade ausgeführte Methode (in der auch das `self` steht) definiert ist, denn das kann ja, aufgrund von Vererbung, durchaus eine Superklasse sein. Das hat eine fundamentale Auswirkung: Die zu einer an `self` geschickten Nachricht passende Methode ist nicht automatisch die, die in derselben Klasse definiert ist, sondern kann durchaus in einer ihrer Subklassen gefunden werden, nämlich dann, wenn die aufrufende Methode selbst erst im Rahmen der Suche in der Kette der Superklassen gefunden wurde. Konkret bedeutet diese (bereits in Abschnitt 10.3 im Kontext abstrakter Klassen beschriebene) sog. *offene Rekursion*, dass das Ergebnis des Ausdrucks

**offene Rekursion
zum Zweiten**

⁴¹ Nicht umsonst ist das heute unter dem Namen JUNIT am besten bekannte Unit-test-Framework zuerst für SMALLTALK entwickelt worden — in SMALLTALK ist Testen die einzige Möglichkeit, Fehler in einem Programm vor seiner Auslieferung zu finden.



bei vorliegenden Klassendefinitionen

<u>Klasse</u>	Super
<u>Instanzmethoden</u>	

```

456 sagMirWasDuBist
  ^ self selbstauskunft

458 selbstauskunft
  ^ 'ich bin Super'

```

sowie

<u>Klasse</u>	Sub
<u>Superklasse</u>	Super
<u>Instanzmethoden</u>	

```

460 selbstauskunft
  ^ 'ich bin leider nur Sub'

```

davon abhängt, von welcher Klasse das Empfängerobjekt **fremder** eine Instanz ist. So liefert

```

462 fremder := Super new.
463 fremder sagMirWasDuBist

```

„ich bin super“,

```

464 fremder := Sub new.
465 fremder sagMirWasDuBist

```

hingegen „ich bin leider nur sub“. Man beachte, dass Vererbung tatsächlich eine Kopieren- und-einfügen-Semantik hat, wie in Abschnitt 11.2 bereits nahegelegt: Wenn man die Implementierung von **sagMirWasDuBist** aus **Super** in **Sub** wiederholt hätte, hätte man das-selbe Ergebnis erzielt.

Während offene Rekursion im gegebenen Beispiel durchaus erwünscht ist und ihr Effekt wohl auch den Erwartungen der Programmiererin entspricht, ergeben sich doch immer wieder Konstellationen, in denen man unangenehm überrascht wird. Das Problem ist unter dem Namen *Fragile-base-class-Problem* bekanntgeworden; es wird in Kapitel 55 (Kurseinheit 6) ausführlicher behandelt.

**Fragile-base-class-
Problem**



12.2 Das Einbeziehen überschriebener Methoden: Nachrichten an super

Nicht selten will eine *überschreibende* Methode die überschriebene nicht komplett ersetzen, sondern lediglich modifizieren. Dies ist z. B. regelmäßig bei den als Konstruktoren fungierenden Klassenmethoden `new` und `new:`: der Fall: Selbst wenn sie überschrieben werden, müssen sie doch das grundlegende Verhalten beibehalten, also neue Instanzen der Klasse zurückgeben. Dies geschieht am sinnvollsten, indem aus der überschreibenden Methode die überschriebene Definition aufgerufen und um die gewünschten zusätzlichen Ausdrücke ergänzt wird. Nur leider ist diese nicht mehr sichtbar — sie wurde ja gerade überschrieben.

Für diesen Zweck verfügt SMALLTALK über eine weitere Pseudovariable, „super“ genannt. Die Verwendung von `super` als Nachrichtenempfänger in einer Methodendefinition bewirkt, dass mit der Suche nach der zur Nachricht passenden, „aufgerufenen“ Methodendefinition in der (direkten) Superklasse der Klasse, in der sich der aufrufende Ausdruck (die aufrufende Methode) befindet, begonnen wird.

Man beachte, dass die Suche anders als bei `self` unabhängig von der Klasse des Objekts ist, für das `super` steht: Obwohl `super` genau wie `self` als Objekt stets den aktuellen Nachrichtenempfänger bezeichnet, bewirkt `super` immer eine von der Klasse des konkreten Empfängerobjekts losgelöste Suche, die eben mit der Superklasse der Klasse, in der `super` verwendet wird, beginnt und nicht etwa mit der Superklasse der Klasse, von der das (durch `super` bezeichnete) Empfängerobjekt eine direkte Instanz ist.

Bedeutung von
super ist
unabhängig von der
Klasse des
Empfängers

Selbsttestaufgabe 12.2

Probieren Sie aus, was passiert, wenn Sie in der Klasse `Sub` von oben folgende Methode hinzufügen:

466 **sagMirWerDuBist**
467 ^ super selbstauskunft

12.3 Double dispatch

Ihnen ist vielleicht aufgefallen, dass im oben beschriebenen Verfahren zum Auffinden der auszuführenden Methode nur das Empfängerobjekt, jedoch nicht die Parameterobjekte berücksichtigt wurden. Das erscheint zunächst natürlich. Manchmal hängt jedoch die Auswahl einer geeigneten Methode auch davon ab.

Typische Fälle, in denen auch die tatsächlichen Parameter eine Rolle bei der Methodenauswahl spielen, sind arithmetische Operatoren wie `+`, `-`, `*` und `/`. Diese sind nämlich sowohl für Ganzzahlen als auch für Brüche und Gleitkommazahlen definiert, wobei die Implementierung einer Operation

wenn Parameter-
objekte über
Methodenauswahl
entscheiden sollen



davon abhängt, welcher Art die Operanden sind. Nehmen wir beispielsweise an, es gäbe zwei primitive Methoden für die Addition, und zwar eine effiziente für die Integer-Addition (**IAdd**) und eine weniger effiziente für die Float-Addition (**FAdd**), und man möchte Additionen für beliebige Kombinationen von Summanden möglichst effizient durchführen können. Dann kommt man vielleicht auf die folgende Tabelle von Zuordnungen:

		Parameter	
		Integer	Float
Empfänger	+	Integer	IAdd FAdd
		Float	FAdd FAdd

Während die Unterscheidung nach Empfängerobjekten vom dynamischen Binden und damit dem Laufzeitsystem vorgenommen wird, bleibt die Frage, wie man die Unterscheidung nach den Parameterobjekten vornimmt: Zumindest die Implementation der Addition in der Klasse **Integer** muss ja danach unterscheiden, ob der Parameter auch ein Integer oder vielleicht ein Float ist. Anstatt nun diese *Fallunterscheidung* (mittels entsprechender Methoden **isInteger** bzw. **isFloat**) explizit zu machen, kann man sich eines einfachen Tricks bedienen: Man ruft im Rumpf einer Methode dieselbe Methode einfach noch einmal auf und vertauscht dabei Empfänger (**self**) und Parameter. Damit es dabei nicht zu unendlichen Rekursionen kommt, kodiert man die Klasse des Empfängers im Nachrichtenselektor der neu aufgerufenen Methode⁴², also z. B. **plusFloat**: anstelle von nur **plus**:. Das Ergebnis sieht dann wie folgt aus:

**Umsetzung durch
erneuten
Methodenauftrag mit
vertauschten
Empfänger- und
Parameterobjekten**

Klasse | **Integer**

Instanzmethoden |

```
468 plus: aNumber
469   ^ aNumber plusInteger: self

470 plusInteger: anInteger
471   <primitive: IAdd>
```

Klasse | **Float**

Instanzmethoden |

```
472 plus: aNumber
473   ^ <primitive: FAdd>

474 plusInteger: anInteger
475   ^ self plus: anInteger
```

⁴² In Sprachen wie JAVA, in denen das *Double dispatch* auch gebräuchlich ist, ist das nicht notwendig, da es in ihnen zur Differenzierung von gleichnamigen Methoden das sog. *Überladen* gibt.



Diese Technik, nämlich eine Methode gleicher Bedeutung unter Vertauschung von Sender und Empfänger aufzurufen, nennt man **Double dispatch**, und zwar, weil die dynamische Bindung (auch *Method* oder *Message dispatching* genannt) zweimal, und zwar unmittelbar hintereinander, erfolgt. Etwas ähnliches haben Sie bei der Implementierung von `+` in `Integer` in Abschnitt 4.3.7 (Kurseinheit 1, Zeile 154) schon gesehen. Die Technik des Double dispatch wurde übrigens von DAN INGALLS am Beispiel von SMALLTALK erstmals beschrieben; sie findet auch in anderen Sprachen mit *Single dispatch* (wie JAVA und C#) verbreitet Anwendung. Double dispatch wird in Sprachen, bei denen bei der (dynamischen) Methodenauswahl von Haus aus die Parametertypen mit berücksichtigt werden (die sog. *Multi-dispatch-Sprachen*), naturgemäß nicht benötigt.



13 Programmieren mit Collections

In Kapitel 2 von Kurseinheit 1 waren wir bereits auf $:n$ -Beziehungen eingegangen, die logisch gleichberechtigt neben $:1$ -Beziehungen stehen, die aber in der Umsetzung besonderer Mechanismen bedürfen. Als Basis der Umsetzung hatten Sie bereits Zwischenobjekte kennengelernt, die über ihre indizierten Instanzvariablen solche Beziehungen — wenn auch nur indirekt — herstellen können. Tatsächlich könnte man, wenn man sich der Häufigkeit des Vorkommens von $:n$ -Beziehungen in der Programmierung bewusst ist, vermuten, dass indizierte Instanzvariablen speziell für diesen Zweck eingeführt wurden. Auf den ersten Blick bedauerlich ist nur, dass dafür eben diese Zwischenobjekte notwendig sind.

Es ergibt sich aus diesem Umstand aber auch ein entscheidender Vorteil.

Da auch diese Zwischenobjekte Instanzen von Klassen sein müssen, ist es

:n-Beziehungen mit Verhalten

möglich, verschiedene Arten von $:n$ -Beziehungen zu definieren und diese jeweils mit Verhalten zu versehen, das speziell auf die Art der Beziehung bezogen ist. So ist es beispielsweise möglich, $:n$ -Beziehungen zu definieren, deren Elemente (die in Beziehung stehenden Objekte) jeweils nur einmal darin vorkommen dürfen (mengenwertige Beziehungen) oder nach einem bestimmten Kriterium sortiert sind. Auch können Operationen wie das Hinzufügen oder Entfernen von Objekten zu einer Beziehung, die bei $:1$ -Beziehungen über die Zuweisung zu einer Instanzvariable erfolgen (das Entfernen durch Zuweisung von `nil`), beliebig ausgestaltet werden, um beispielsweise die Mengenwertigkeit oder die Sortierung zu erhalten.

Besonders attraktiv ist jedoch die in SMALLTALK bestehende Möglichkeit, eigene Kontrollstrukturen für $:n$ -Beziehungen zu spezifizieren. Die bereits vorhandenen durften Sie ja schon in Abschnitt 4.6.4 kennenlernen; hier kommt hinzu, dass die Standarditeratoren je nach Art der Beziehung unterschiedliche Eigenschaften haben. Außerdem ist es natürlich möglich, mit eigenen Arten von Beziehungen auch spezielle, nur für diese Beziehungen benötigte Kontrollstrukturen zu spezifizieren. Doch zunächst zur Pflege von solchen Beziehungen.

spezielle Kontrollstrukturen für :n-Beziehungen



13.1 Pflegen von :n-Beziehungen

Um :n-Beziehungen zu pflegen, also um Objekte zu einer Beziehung hinzuzufügen und wieder zu entfernen, sieht SMALLTALK standardmäßig die Methoden `add:` und `remove:` vor, die beide jeweils das Argumentobjekt zurückliefern. Beide sind in der abstrakten Klasse `Collection` definiert, die Wurzel einer Hierarchie von Klassen, den *Collection-Klassen*, ist, die allesamt der Verwirklichung von :n-Beziehungen dienen. Unsere Zwischenobjekte, die diese Beziehungen repräsentieren, sind also alle indirekte Instanzen von `Collection`.

Die Methoden `add:` und `remove:` bleiben zunächst (in `Collection`) abstrakt:

Methoden add: und remove:

```
476 add: anObject
477   ^ self implementedBySubclass

478 remove: anObject
479   ^ self
480     remove: anObject
481     ifAbsent: [self errorAbsentObject]

482 remove: anObject ifAbsent: anExceptionBlock
483   ^ self implementedBySubclass
```

Da sie von der tatsächlichen Realisierung einer Collection abhängen, können sie erst in den entsprechenden Subklassen (durch *Überschreiben*) realisiert werden.

Beim Entfernen eines Objektes aus einer Collection⁴³ mittels `remove:` gibt es zwei Sonderfälle zu berücksichtigen: Das Objekt ist nicht vorhanden oder das Objekt ist mehrfach vorhanden. Im ersten Fall wird ein Fehler gemeldet, während im zweiten nur ein Vorkommen des Objekts aus der Collection entfernt wird (das erste, wie auch immer die Reihenfolge festgelegt ist). Da es immer vorkommen kann, dass ein zu entfernendes Objekt gar nicht vorhanden ist, und ein entsprechender vorheriger Test auf Vorhandensein (s. u.) wieder so eine stereotype Handlung ist, bietet SMALLTALK eine Variante von `remove:`, die einem genau das erspart: `remove: anObject ifAbsent: anExceptionBlock`. Sollte das zu entfernende Objekt fehlen, wird stattdessen `anExceptionBlock` ausgeführt und dessen Ergebnis zurückgeliefert. Will man, dass beim Versuch, ein nicht vorhandenes Objekt zu entfernen, nichts passiert, so gibt man einfach den leeren Block `[]` für `anExceptionBlock` an. Sollen mehrere Objekte auf einmal einer Beziehung hinzugefügt bzw. daraus entfernt werden, so stehen hierfür die Methoden `addAll: aCollection` bzw. `removeAll: aCollection` zur Verfügung, die jeweils eine Collection als Parameter erwarten.

Sonderfälle bei remove:

⁴³ Wir reden im folgenden von Collections anstelle von :n-Beziehungen und tun dies in dem Bewusstsein, dass es sich bei den entsprechenden Instanzen prinzipiell um eigenständige Objekte handelt, die hier lediglich die Funktion eines Zwischenobjektes haben, also dem Zweck der Realisierung der Beziehungen dienen.



Subklassen von Collection müssen also die Methoden `add:` und `remove:ifAbsent:` überschreiben. Dabei offenbart sich gleich ein Charakterzug SMALLTALKS: Da seine Klassenhierarchie keine Generalisierungshierarchie ist, kommt es vor, dass Subklassen die Methoden `add:`, `remove:` und `remove:ifAbsent:` löschen. Während beispielsweise in der Klasse `OrderedCollection` `add:` und `remove:ifAbsent:` mit

```

484 add: anObject
485   endPosition = contents size
486   ifTrue: [self putSpaceAtEnd].
487   endPosition := endPosition + 1.
488   contents at: endPosition put: anObject.
489   ^ anObject

490 remove: anObject ifAbsent: aBlock
491   | index |
492   index := startPosition.
493   [index <= endPosition]
494     whileTrue: [
495       anObject = (contents at: index)
496       ifTrue: [
497         self removeIndex: index - startPosition + 1.
498         ^ anObject].
499       index := index + 1].
500   ^ aBlock value

```

überschrieben werden, werden sie in der Klasse `FixedSizeCollection`, die ebenfalls eine Subklasse von Collection ist, gelöscht:

```

501 add: anObject
502   ^ self invalidMessage

503 remove: anObject ifAbsent: aBlock
504   ^ self invalidMessage

```

Die Methoden `add:` und `remove:` werden durch die Methoden `addAll:` und `removeAll:` komplettiert; die Implementierung von `addAll:` können Sie den Zeilen 427–431 oben (Abschnitt 10.3) entnehmen, `removeAll:` verläuft im Prinzip analog (warum SMALLTALK EXPRESS hier eine Kopie zurückgibt, weiß ich nicht). Die Methode `addAll:` wird dazu benutzt, eine Collection in eine andere zu konvertieren:

```

505 asBag
506   ^ (Bag new)
507     addAll: self;
508     yourself

509 asOrderedCollection
510   ^ (OrderedCollection new: self size)
511     addAll: self;
512     yourself

```



Dabei ist **addAll**: nur einmal, nämlich in **Collection**, definiert. Man beachte, dass dabei ein Objekt nicht seine Klasse wechselt, sondern lediglich der Inhalt einer Collection in eine neue übertragen wird. Diese Übertragung ist immer dann sinnvoll, wenn die Klasse der neuen Collection Eigenschaften hat, die man gern nutzen möchte. Ein Beispiel hierfür finden Sie in Zeile 525 unten. Die Nachricht **yourself** (von **Object** geerbt) liefert übrigens ihren Empfänger zurück; sie wird am Ende von kaskadierten Nachrichtenausdrücken in Return-Anweisungen verwendet, um den Empfänger zurückzuliefern.

Zum Pflegen seiner Beziehungen ist es manchmal vorteilhaft, zu wissen, mit wie vielen Objekten man in Beziehung steht und mit welchen. Die Klasse **Collection** sieht dafür die Methoden **size**, **isEmpty** und **notEmpty**, **includes**: sowie **occurrencesOf**: vor, die jeweils die nahegelegte Bedeutung haben.

Weitere nützliche Methoden zum Pflegen von :n-Beziehungen

13.2 :n-Beziehungen mit besonderen Eigenschaften

Die Klasse **Collection** ist wie gesagt abstrakt. SMALLTALK sieht nun eine ganze Hierarchie von spezielleren, instanzierbaren (konkreten) **Collection**-Klassen vor, die für die unterschiedlichsten Zwecke eingesetzt werden können. Darunter sind so offensichtliche wie **Set** (für ungeordnete Collections, in denen jedes Element höchstens einmal vorkommen darf, also Mengen) und **Bag** (für solche, in denen die letzte Einschränkung aufgehoben ist). **Set** und **Bag** haben (neben der mangelnden Ordnung ihrer Elemente) gemein, dass die Elemente in beiden nicht über einen Index zugreifbar sind. Im Gegensatz dazu stehen geordnete Collections (Klasse **SequenceableCollection** oder **IndexedCollection**, je nach System), in denen das *i*-te Element eindeutig bestimmt ist und die entsprechend die Methoden **at**: und **at :put** : implementieren (genaugenommen überschreiben, denn diese Methoden sind ja für alle Objekte, die über indizierte Instanzvariablen verfügen, schon definiert, und werden lediglich für ungeordnete Collections wieder gelöscht). Aber auch ungeordnete Collections (in denen keine Reihenfolge festgelegt ist) können indiziert sein: In Objekten der Klasse **Dictionary** wird jedes Element unter einem Schlüssel, der selbst wieder ein Objekt sein kann, gespeichert. Die dazugehörigen Methoden heißen wiederum **at**: und **at :put** :, erlauben aber Objekte anderer Klassen als **Integer** als Indizes.

13.2.1 Dictionaries

Dictionaries repräsentieren sog. qualifizierte Beziehungen, das sind solche, bei denen jedes Element der Beziehung durch einen Qualifizierer eindeutig bestimmt wird. Der Qualifizierer heißt auch **Schlüssel** (engl. key; vergleichbar mit dem Primärschlüssel relationaler Datenbanken), das qualifizierte Element der Beziehung nennt man auch **Wert** (engl. value). Ein Element einer qualifizierten Beziehung besteht also gewissermaßen aus einer Assoziation eines Schlüssels mit einem Wert. Der Clou an der Implementierung von Dictionaries ist, dass man Werte unter ihren Schlüsseln extrem schnell (im Idealfall ohne jede Suche) auffinden kann. Das wird heute fast immer über sog. Hashing erreicht.



Die Klasse **Dictionary** hat für die Programmierung besondere Bedeutung: Sie realisiert sog. Assoziativspeicher, also Speicher, bei dem auf eine Speicherzelle nicht durch Angabe einer Speicheradresse, sondern durch Assoziation mit dem Inhalt zugegriffen wird. Sie wird im SMALLTALK-System selbst häufig verwendet. So werden z. B. Methoden in Dictionaries hinterlegt (wobei der Nachrichtenselektor die Rolle des Schlüssels spielt und als Wertobjekt die kompilierte Methode gespeichert ist). Aber auch andere Arten von Collections lassen sich mit Hilfe von Dictionaries sehr einfach realisieren: So kann man die Klasse **Bag** beispielsweise wie folgt implementieren:

<u>Klasse</u>	Bag
<u>Superklasse</u>	Collection
<u>Klassenmethoden</u>	
513 new	
514 ^ self basicNew initialize	
<u>benannte Instanzvariablen</u>	elements
<u>indizierte Instanzvariablen</u>	nein
<u>Instanzmethoden</u>	
515 initialize	
516 elements := Dictionary new	
517 add: anObject	
518 elements	
519 at: anObject	
520 put: (self occurrencesOf: anObject) + 1.	
521 ^ anObject	
522 occurrencesOf: anObject	
523 ^ elements at: anObject ifAbsent: 0	
524 ...	

Dabei wird einfach die Anzahl der Vorkommen eines Elements (repräsentiert durch den formalen Parameter **anObject**) der Bag, solange diese nicht Null ist, in einem Dictionary unter dem Element als Schlüssel gespeichert.

Man beachte hierbei, dass **Bag** die Klasse **Dictionary** nutzt, ohne von ihr zu erben. Stattdessen hält sich jede Instanz von **Bag** eine Instanz von **Dictionary** als Sklavin, die für sie den Dienst verrichtet. Man spricht hier auch von einer **Delegation**⁴⁴; da

Delegation

⁴⁴ Tatsächlich würde manche es lieber sehen, wenn man hier von **Forwarding** und nicht von Delegation sprechen würde. Die Delegation übernimmt nämlich in den prototypenbasierten objektorientierten Programmiersprachen tatsächlich die Funktion der Vererbung, wobei insbesondere die Pseudovariable **self** in den delegierten Methoden die gleiche Bedeutung wie bei einer geerbten Me-



die Delegation auf Instanzebene stattfindet und zudem dynamisch (also nachdem eine Instanz erzeugt wurde) eingerichtet werden kann und da sie zudem von Fragen der Generalisierung/Spezialisierung völlig befreit ist, erfreut sie sich in der objektorientierten Programmierung großer Beliebtheit.

Je länger Sie in SMALLTALK programmieren, desto häufiger werden Sie feststellen, dass Sie durch Verwendung eines Dictionaries Ihren Code deutlich vereinfachen können. Tatsächlich erlauben es Dictionaries (bzw.

Dictionaries als
universelle
Datenstruktur

der von ihnen realisierte Assoziativspeicher), Assoziationsketten, die Grundlage vieler menschlicher Denkprozesse sind, direkt in einem Programm nachzubilden. Fragen Sie sich also, wann immer Sie es mit einer Menge von Objekten zu tun haben, wie Sie auf die Elemente der Menge zugreifen wollen; wenn dies über einen Schlüssel erfolgt, dann ist **Dictionary** die Klasse Ihrer Wahl.

Es darf übrigens der Schlüssel eines in einem Dictionary gespeicherten Objekts ruhig ein Attribut (der Inhalt einer Instanzvariable) des Objekts sein; dies kommt sogar recht häufig vor. Beispielsweise wird man Personen in einem Dictionary unter ihrem Nachnamen oder einer Ausweisnummer speichern. Allerdings sollte dieses Attribut dann unveränderlich sein, da das Objekt nach einer Änderung des Attributs immer noch unter dem alten Attributwert als Schlüssel gespeichert ist und nur unter diesem wiedergefunden wird.

Vorsicht Falle!

13.2.2 Sortierte Collections

Eine weitere nützliche Collection-Klasse wird durch **SortedCollection** implementiert. Es handelt sich dabei um eine Subklasse von **OrderedCollection**, bei der die Reihenfolge der Elemente nicht von außen, also durch die Angabe eines Indexes oder die Reihenfolge der Einfügung, festgelegt wird, sondern von innen, genauer durch eine Qualität der eingefügten Objekte. Zwischenobjekte der Klasse **SortedCollection** setzt man ein, wenn man die in Beziehung stehenden Objekte in einer bestimmten Reihenfolge stehen wissen möchte, wie z. B. die Kinder einer Person in der Namensfolge, und zwar unabhängig davon, in welcher Reihenfolge sie der Collection hinzugefügt wurden. Voraussetzung dafür, dass die Elemente einer **SortedCollection** sortiert werden können, ist, dass sie sich vergleichen lassen, dass also die (binäre) Methode `<=` (für kleiner gleich) darauf definiert ist. So liefert beispielsweise

525 `#(3 2 1) asSortedCollection asArray`

thode hat. Dies ist beim Forwarding (oder eben der „Delegation“) der klassenbasierten objektorientierten Programmierung nicht der Fall: `self` in der Methode `at:put:` der Klasse **Dictionary** bezieht sich nie auf ein Objekt der Klasse **Bag**.



mit #(1 2 3) das gewünschte Ergebnis.⁴⁵

Wenn die Elemente, die in eine sortierte Collection eingefügt werden sollen, keine Größen sind, also insbesondere den Vergleich `<=` nicht implementieren, dann ist es immer noch möglich, für eine neue Instanz einer `SortedCollection` einen sog. Sortierblock zu spezifizieren, der zwei *formal Parameter* hat und dessen Auswertung zurückliefert, ob der erste tatsächliche Parameter kleiner oder gleich dem zweiten ist. Tatsächlich wird, falls man bei der Erzeugung keinen Sortierblock angibt, ein Standardsortierblock angenommen:

Sortierblöcke

```
Klasse | SortedCollection
Superklasse | OrderedCollection
Klassenmethoden | 

526 sortBlock: aBlock
527   "Answer aSortedCollection which will
528     sort in the order defined by aBlock."
529   ^ (super new: 10) sortBlock: aBlock

530 new: anInteger
531   "Answer a SortedCollection capable of
532     holding anInteger number of elements
533     which will sort in ascending order."
534   ^ (super new: anInteger) sortBlock: [ :a :b | a <= b]

benannte Instanzvariablen | sortBlock
indizierte Instanzvariablen | nein
Instanzmethoden | 

535 sortBlock: aBlock
536   "Answer the receiver. Set the sort block for
537     the receiver to aBlock and resort the receiver."
538   sortBlock := aBlock.
539   self reSort

540 ...
```

Vorsicht Falle!

⁴⁵ Analog dazu gibt es noch eine ganze Reihe anderer Konvertierungsmethoden, mit deren Hilfe eine Menge von Objekten aus einer Collection in eine andere übertragen werden kann, wobei die Eigenschaften der Ziel-Collection berücksichtigt werden, so z. B. `asSet`, das doppelte Elemente entfernt. Sie sind allesamt (und analog zu `asBag` und `asOrderedCollection` oben) in `Collection` implementiert. Besonders interessant sind natürlich Konvertierungen in Collections, die stärkere Bedingungen stellen, also z. B. eben `asSet` und `asSortedCollection`.



Man beachte jedoch, dass eine nachträgliche Änderung der Attributwerte, die zum Vergleich der Objekte für die Sortierung herangezogen wurden, keine automatische Änderung der Reihenfolge bewirkt, selbst wenn dies eigentlich notwendig würde.

13.2.3 Arrays

Nicht zuletzt werden auch ganz banale Arrays häufig verwendet, insbesondere wegen der (bereits in Abschnitt 1.2 vorgestellten) Möglichkeit der einfachen *literalen Definition*. So kann man ohne viel Aufwand über die Elemente einer beliebigen, ad hoc spezifizierten Aufzählung iterieren:

```
541 #(1 'abc' true) collect: [ :element | ... ]
```

beispielsweise weist dem Laufparameter des Blocks, `element`, nacheinander die Elemente des literalen Arrays zu.

Der wesentliche Nachteil von Arrays ist, dass ihre Größe beschränkt ist. Benötigt man eine geordnete Collection, die beliebig wachsen kann, der also am Anfang, am Ende oder an einer beliebigen Position dazwischen Elemente hinzugefügt werden können, dann kann man auf Instanzen der Klasse `OrderedCollection` zurückgreifen. Diese eignen sich aufgrund des angebotenen Methodensatzes, ihres *Protokolls*, speziell für die Implementierung von Stapeln (Stacks) und Puffern (Queues).

**beschränkte Größe
von Arrays**

13.3 Collections für andere Zwecke

Nicht alle Collections dienen der Umsetzung von $:n$ -Beziehungen. Ein gutes Beispiel gibt die Klasse `Interval`.

Bei Instanzen der Klasse `Interval` handelt es sich um endliche arithmetische Folgen, also um beschränkte Folgen von Zahlen, die alle denselben Abstand zueinander haben. Die Elemente einer solchen Collection müssen deswegen nicht gespeichert, sondern können berechnet werden. Die Spezifikation eines Intervalls umfasst seinen Anfangs- und seinen Endwert sowie die Schrittweite, die auch negativ sein darf.

**Intervalle als
Collections**

```
542 (Interval from: 5 to: 1 by: -2)
```

erzeugt ein Intervall, das die Zahlen 5, 3 und 1 enthält. Intervalle dienen vor allem dem Zweck, sog. For-Schleifen zu emulieren (s. Abschnitt 4.6.3 in Kurseinheit 1):

```
543 (Interval from: 5 to: 1 by: -2) do: [ :i | ... ]
```

etwa bewirkt, dass dem Laufparameter `i` nacheinander die Werte 5, 3 und 1 zugewiesen werden. Die zweiparametrische Form

```
544 (Interval from: 1 to: 10) do: [ :i | ... ]
```



nimmt hingegen eine Standardschrittweite von 1 an. Für noch mehr Komfort ist in der Klasse **Number** eine Methode **to:by:** vorgesehen, die ein entsprechendes Intervall zurückliefert:

```
545 to: eNumber by: iNumber
546   ^ Interval from: self to: eNumber by: iNumber
```

erlaubt, statt Zeile 544

```
547 (5 to: 1 by: -2) do: [ :i | ... ]
```

zu schreiben. Um der geschätzten Programmiererin auch noch die Klammern zu ersparen, wurde gleich noch die Methode **to:by:do:** hinzugefügt, die daraus

```
548 5 to: 1 by: -2 do: [ :i | ... ]
```

zu machen erlaubt. (Man beachte, dass hier der Iterator in der Klasse **Number** und nicht in einer Collection wie **Interval** definiert wurde.) Wie man sieht, ist es in SMALLTALK möglich, ohne großen Aufwand neue Ausdrucksformen hinzuzufügen, ohne dazu (wie in den meisten anderen Sprachen notwendig) die Syntax ändern zu müssen.

Selbsttestaufgabe 13.1

Sie finden folgendes Codefragment vor:

```
549 1 to: a size do: [ :i | (a at: i) doSomething ]
```

Kritisieren Sie es!

Zu guter Letzt sind auch die Klassen **String** und **Symbol** Collections, und zwar genauer geordnete Collections fester Größe (wie Arrays), deren Inhalt jedoch auf Zeichen (Instanzen der Klasse **Character**) beschränkt ist. Strings sehen neben der Möglichkeit des Vergleichs (mittels der Operatoren `<`, `<=`, `>`, `>=`) auch noch ein Pattern matching (Methode **match: aString**) sowie spezielle Operatoren zur Behandlung von Groß-/Kleinschreibung und eine Konversion in Literale vor. (Die Klasse **Symbol** ist übrigens eine Subklasse der Klasse **String**; sie erbt damit alle Methoden von **String**.) Auf die Möglichkeiten der literalen Repräsentation von Strings und Symbolen wurde bereits in Kurseinheit 1, Abschnitt 1.2, eingegangen.

String und Symbol als Collections

Selbsttestaufgabe 13.2

Überlegen Sie sich, wie Sie das Case- (oder Switch-)Statement in SMALLTALK simulieren würden, und gehen Sie auf die Einschränkungen ein, die Sie dazu machen müssen.



14 Verhalten für alle Objekte

In der Klasse `Object` ist das Protokoll definiert, das allen Objekten gemein ist, d. h., für das alle Klassen Methodendefinitionen haben, und zwar entweder eigene oder geerbte. `Object` gibt zu diesem Zweck Standardimplementierungen vor, die von den meisten Objekten direkt übernommen werden können (und nur von den wenigsten überschrieben werden müssen). Dazu zählen z. B. die bereits mehrfach erwähnten `isNil` und `notNil` (die nur von `UndefinedObject` überschrieben werden) sowie zahlreiche weitere Typtests (`isInteger`, `isFloat` usw.). Daneben gibt es auch noch eine ganze Reihe anderer Methoden, die zu kennen es sich lohnt.

14.1 Kopieren von Objekten

In Abschnitt 7.3 hatten wir ja die Instanziierung als den hauptsächlichen Weg kennengelernt, wie neue Instanzen von Klassen, für deren Objekte es keine literale Repräsentation gibt, erzeugt werden. Wir hatten allerdings dort schon auf die Möglichkeit des Klonens/Kopierens hingewiesen. Darauf wollen wir nun wieder zurückkommen.



Die einfachste Form des Kopierens eines Objekts erzeugt ein Objekt gleicher Klasse mit gleichen Variablenbelegungen. Dazu gibt es in SMALLTALK die Methode

flaches Kopieren

```
550 shallowCopy
551     "Answer a copy of the receiver which shares
552         the receiver instance variables."
```

Diese Methode liefert eine neue Instanz der Klasse des Empfängers, die in denselben Beziehungen zu denselben anderen Objekten steht wie das Original. Insbesondere werden die Objekte, die die Instanzvariablen des Originals benennen, nicht selbst kopiert. Deswegen nennt man die Kopie **flach**. Sie erfolgt einfach durch Zuweisung aller Instanzvariablen des Originals an die Instanzvariablen des neuen Objekts, das damit zur Kopie wird. Die Implementierung in SMALLTALK EXPRESS ist die folgende:

```
553 | answer aClass size |
554 aClass := self class.
555 aClass isVariable
556 ifTrue: [
557     size := self basicSize.
558     answer := aClass basicNew: size]
559 ifFalse: [
560     size := 0.
561     answer := aClass basicNew].
562 aClass isPointers
563 ifTrue: [
564     1 to: size + aClass instSize do: [ :index |
565         answer instVarAt: index
566             put: (self instVarAt: index)]]
567 ifFalse: [
568     1 to: size do: [ :index |
569         answer basicAt: index
570             put: (self basicAt: index)]].
571 ^ answer
```

isVariable unterscheidet dabei zwischen Klassen mit indizierten Instanzvariablen und solchen ohne; **isPointers** unterscheidet zwischen Klassen mit *zusammengesetzten* Objekten und *atomaren*.

Nun ist eine flache Kopie aber häufig nicht genug. Es gibt daher noch eine zweite Methode

tiefes Kopieren

```
572 deepCopy
573     "Answer a copy of the receiver with shallow
574         copies of each instance variable."
```

Wie der Name nahelegt, unterscheidet sich die Methode **deepCopy** von **shallowCopy** darin, dass auch die in Beziehung stehenden (durch die Instanzvariablen benannten) Objekte kopiert werden. Statt einzelner Objekte wird also ein Objektgeflecht kopiert — die Kopie ist **tief**. Es muss dazu an die beiden tatsächlichen Parameter von **put:** (Zeilen 566 und 570) lediglich eine Nachricht zum Kopieren der Parameter angehängt werden. Dabei ist jedoch



Vorsicht geboten: Wenn es sich dabei ebenfalls um ein tiefes Kopieren handelt, dann kann der Kopiervorgang leicht in eine Endlosrekursion geraten.

Selbsttestaufgabe 14.1

Überlegen Sie, wie Sie ein rekursives tiefes Kopieren technisch in den Griff bekommen können.

Nun ist die Festlegung, ob die Kopien ihrer Instanzen tiefe oder flache sein sollen, gelegentlich ein Charakteristikum der Klasse selbst. Jede Klasse erbt deswegen von `Object` eine Methode `copy`, die standardmäßig (also in `Object`) einfach `shallowCopy` aufruft (warum es nicht `deepCopy` aufruft, sollte klar sein) und die die erbende Klasse entsprechend ihren eigenen Konditionen überschreiben kann. Es ist so möglich, die Kopiertiefe von Objektstrukturen selbst zu bestimmen, indem man `copy` für manche Klassen `deepCopy` aufrufen lässt und das tiefe Kopieren durch Instanzen terminiert, deren Klassen `shallowCopy` aufrufen lassen.

klassenspezifische Kopiertiefe

Manchmal darf bei Kopier- oder Konvertieroperationen kein Objekt des gleichen Typs zurückgegeben werden. In diesen Fällen sollte statt `self class` (Zeile 554) `self species` aufgerufen werden:

Kopiervorgänge mit self species

```
575 species
576   "Answer a class which is similar to (or the same
577     as) the receiver class which can be used for
578     containing derived copies of the receiver."
```

Die Methode `species` war uns schon einmal begegnet, und zwar in Kurseinheit 1, Abschnitt 4.6.4, Zeile 244. Sie gibt standardmäßig die Klasse des Empfängerobjekts zurück und kann überschrieben werden, wenn eine andere Klasse angegeben werden soll. Dies ist z. B. bei der Methode `collect:`, ausgeführt auf einer Instanz von `Interval`, sinnvoll, da `collect:` hier kein Intervall zurückgeben kann. So kann beispielsweise die von

```
579 (Interval from: 1 to: 5) collect: [ :n | n printString ]
```

zurückgegebene Collection von Strings nicht als Intervall dargestellt werden. Entsprechend ist in der Klasse `Interval` die Methode `species` als

```
580 species
581   ^ Array
```

implementiert.

14.2 Reinkarnation von Objekten

Eine der vielleicht interessantesten Methoden SMALLTALKs ist die Methode `become::`:

```
582 become: anObject
```



583 "The receiver takes on the identity of anObject.
584 All the objects that referenced the receiver
585 will now point to anObject."

Sie bewirkt, dass das Empfängerobjekt die Identität des Parameterobjekts annimmt bzw. sie mit ihm tauscht (je nach Dialekt). Das hat u. a. zur Folge, dass alle Variablen, die vor der Ausführung der Methode den Empfänger benannten (genauer: auf das Empfängerobjekt verwiesen), danach den Parameter benennen (auf ihn verweisen).

Eine mögliche Anwendung ist das *Wachsen von Objekten*: Wenn einem Objekt der ihm zur Verfügung gestellte Speicherplatz nicht mehr ausreicht, muss es „umziehen“, d. h., seine Repräsentation im Speicher muss an eine andere Stelle kopiert werden. Da aber alle Referenzen auf das Objekt noch auf die alte Stelle verweisen, legt man am besten die neue Stelle als entsprechend groß dimensioniertes Objekt an (beispielsweise mittels `new:`) und lässt dann das alte Objekt zum neuen werden. So könnte man beispielsweise eine Methode `grow` in der Klasse `ArrayedCollection` wie folgt definieren:

```
586 grow
587     "Answer the receiver expanded in
588     size to accomodate more elements."
589 | size new |
590 size := self size.
591 new := self species new: size + self growSize.
592 new replaceFrom: 1 to: self size with: self.
593 self become: new
```

wachsende Objekte

Eine andere mögliche Anwendung von `become:` ist die Durchführung eines sog. Rollback, wenn also, nachdem an einem Objekt (oder Objektgeflecht) eine Menge von Änderungen durchgeführt worden sind, der ursprüngliche Zustand wiederhergestellt werden soll. Man legt dann einfach vor den Änderungen eine (tiefe) Kopie des Objekts (der Wurzel des Objektgeflechts) an und ersetzt beim Rollback das ursprüngliche (und inzwischen geänderte) Objekt(geflecht) mittels `become:` durch die Kopie.

Rollback mittels `become:`

14.3 Kommunikation mit mehreren: Multicasting

Neben vielen anderen Neuerungen wird SMALLTALK auch das Model-View-Controller-Entwurfsmuster (MVC-Pattern) zugeschrieben, das sich heute noch (auch in Web-Anwendungen) großer Beliebtheit erfreut. Beim MVC-Pattern gibt es verschiedene (An-)Sichten auf ein logisches Modell, und da Änderungen im Modell potentiell alle Sichten betreffen, muss jede Änderung alle Sichten darüber unterrichten. Es ist also eine Eins-zu-viele-Kommunikation erforderlich, die nicht durch den normalen Nachrichtenversand abgedeckt wird.



WIKIPEDIA

Das folgende Protokoll setzt diese Form der Kommunikation in SMALLTALK um; es ist vollständig in SMALLTALK implementiert und sollte Ihnen inzwischen kein Problem mehr bereiten. Beachten Sie, dass `Object` keine *Lazy initialization* seiner Klassenvariable `Dependents` vorsieht; die Methode `initDependents` muss daher bei Erzeugung einer neuen Klasse jeweils einmal aufgerufen werden.



Klasse

Object

Klassenvariablen

Dependents ...

Klassenmethoden

594 initDependents

595 "Initialize the Dependents dictionary to empty."
596 Dependents := IdentityDictionary new

benannte Instanzvariablen

indizierte Instanzvariablen

nein

Instanzmethoden

597 addDependent: anObject

598 "Add anObject to the class variable
599 Dependents of class Object."
600 (Dependents at: self ifAbsent: [
601 Dependents at: self put: OrderedCollection new])
602 add: anObject

603 dependsOn: anObject

604 "Add the receiver to anObject's
605 collection of dependents."
606 anObject addDependent: self

607 dependents

608 "Answer a collection of all
609 dependents of the receiver."
610 ^ Dependents at: self
611 ifAbsent: [^ OrderedCollection new]

612 broadcast: aSymbol

613 "Send the argument aSymbol as a unary
614 message to all of the receiver's dependents."
615 self dependents do: [:dependent | dependent perform: aSymbol]

616 changed

617 "The receiver changed in some general way. Inform all
618 dependents by sending each dependent an update message."
619 self changed: self

620 changed: aParameter

621 "Something has changed related to the dependents
622 of the receiver. Send the 'update: aParameter'
623 message to all the dependents."
624 (Dependents at: self ifAbsent: [#()]) do: [:dependent |
625 dependent update: aParameter]

626 update: aParameter

627 "An object on whom the receiver is dependent
628 has changed. The receiver updates its status
629 accordingly (the default behavior is to do nothing).
630 The argument aParameter usually identifies the
631 kind of update."



```
632     "default do nothing"  
  
633 release  
634     "Discard all dependents of  
635         the receiver, if any."  
636     Dependents removeKey: self ifAbsent: []  
  
637 ...
```

14.4 Selbstdarstellung

Wir hatten bereits ausgenutzt, dass alle Objekte SMALLTALKs eine (mehr oder weniger aussagekräftige) String-Repräsentation besitzen: Die Methode

```
638 printString  
639     "Answer a String that is an ASCII representation  
640         of the receiver."
```

gibt eine Darstellung des Objekts als String zurück. Dies ist für Ausgaben auf dem Transcript interessant, aber auch für die Inspektion von Objekten und für das Debugging, bei denen sich die Objekte unter Verwendung dieser Methode der Betrachterin präsentieren.

Die Methode **inspect**

Inspektion

```
641 inspect  
642     "Open an inspector window on the receiver."
```

startet auf dem Empfänger einen Inspektor und gibt den Empfänger zurück. Dies ist nützlich, wenn man ein Zwischenergebnis eines Ausdrucks inspizieren möchte, ohne den Ausdruck dazu in zwei aufteilen zu wollen — man fügt einfach **inspect** an der Stelle des Ausdrucks, an der das zu inspizierende Objekt gewonnen wurde, ein.

Die Methode

Ausgabe auf einem Strom

```
643 storeOn: aStream  
644     "Append the ASCII representation of the  
645         receiver to aStream from which the  
646         receiver can be reinstated."
```

erlaubt, ein Objekt so auf einen Ausgabestrom (s. u.) zu schreiben, dass es daraus rekonstruiert werden kann. Dabei wird keine binäre, sondern eine textuelle Repräsentation verwendet. So schreibt beispielsweise SQUEAK bei Auswertung von **Time noon storeOn: aStream** die Zeichenfolge '12:00 pm' **asTime** auf den durch **aStream** bezeichneten Ausgabestrom und (**Interval from: 1 to: 5**) die Zeichenfolge '(1 to: 5)'.



15 Ein- und Ausgabeströme

Ein- und Ausgabeströme spielen in der konventionellen (objektorientierten) Programmierung eine wichtige Rolle, da über sie Eingaben in und Ausgaben aus dem System erfolgen, und zwar sowohl von/zu der Benutzerin als auch vom/zum Dateisystem. Nun ist SMALLTALK aber als fensterbasiertes, grafisches System konzipiert, das der zeilenorientierten und textbasierten Ein- und Ausgabe der damals vorherrschenden Programme eine Alternative gegenüberstellen wollte. Zudem ist auch eine Speicherung permanenter Daten in Dateien nicht nötig, da mit dem Image alle Objekte dauerhaft gespeichert werden. Da ist es nur konsequent, dass die Ein- und Ausgabe über Streams wenig Gewicht hat.

In SMALLTALK haben Streams somit zunächst auch eine andere Aufgabe: Sie erlauben eine Form des Zugriffs auf *Collections*, die das Collection-Protokoll nicht bieten kann, nämlich

Streams als Iteratoren mit besonderen Eigenschaften

- den sequentiellen Zugriff auf einzelne Elemente in beliebigen zeitlichen Abständen (bei den Iteratoren wird immer in einem Schritt, oder in einer Anweisung, über die ganze Collection iteriert) sowie
- den gleichzeitigen bzw. zeitlich abwechselnden Zugriff auf (die Elemente einer) Collection durch mehrere andere Objekte.

Um dies umzusetzen, braucht man Positionszeiger in eine Collection hinein, und genau die zu liefern ist die Funktion von Streams.

Streams werden zunächst immer auf einer Collection erzeugt, deren Inhalt Basis des Streams ist. Die Erzeugung erfolgt mittels der Klassenmethode `on:`, die als Parameter eine Collection erhält. Das Basisprotokoll auf Instanzebene enthält die folgenden Methoden:

Implementierung der Klasse Stream

```
647 contents
648   "Answer the collection over which
649     the receiver is streaming."
650 next
651   "Answer the next object accessible by the receiver
652     and advance the stream position. Report an error
653     if the receiver stream is positioned at end."
654 nextPut: anObject
655   "Write anObject to the receiver stream.
656     Answer anObject."
657 next: anInteger
658   "Answer the next anInteger number of items from
659     the receiver, returned in a collection of the
660     same species as the collection being streamed
661     over."
662 nextPutAll: aCollection
```



```

663     "Write each of the objects in aCollection to the
664     receiver stream. Answer aCollection."
665 nextMatchFor: anObject
666     "Access the next object in the receiver. Answer
667     true if it equals anObject, else answer false."
668 skip: anInteger
669     "Increment the position of the
670     receiver by anInteger."
671 skipTo: anObject
672     "Advance the receiver position beyond the next
673     occurrence of anObject, or if none, to the end of
674     stream. Answer true if anObject occurred, else
675     answer false."
676 atEnd
677     "Answer true if the receiver is
678     positioned at the end (beyond
679     the last object), else answer
680     false."

```

Für frei positionierbare Streams kommt noch das Protokoll zur Änderung des Zeigers hinzu:

```

681 position
682     "Answer the current receiver stream position."
683 position: anInteger
684     "Set the receiver stream position to anInteger.
685     Report an error if anInteger is outside the
686     bounds of the receiver collection."
687 reset
688     "Position the receiver stream to the beginning."
689 setToEnd
690     "Set the position of the receiver stream to
691     the end."
692 peek
693     "Answer the next object in the receiver stream
694     without advancing the stream position. If the
695     stream is positioned at the end, answer nil."

```

Für **peek** ist die freie Positionierbarkeit notwendig, weil man dazu erst das nächste Element anspringen und dann wieder einen Schritt zurückgehen muss.

Da ein Stream (wie eine Collection) eine Menge von Objekten repräsentiert, möchte man darüber (genau wie über eine Collection) iterieren können. Kein Problem:

Iteration über Streams

```

696 do: aBlock
697     "Evaluate aBlock once for each element in the
698     receiver, from the current position to the end."
699     [self atEnd]

```



Außerdem wird natürlich zwischen (nur) lesbaren und schreibbaren Streams unterschieden.

Erst eine weitere Kategorie von Streams operiert nicht auf Collections, sondern auf externen Daten. Dazu gehören insbesondere die File streams. In SMALLTALK-80 wurde mit den Klassen **FileDirectory**, **File** und **FilePage** (die selbst keine Streams sind) ein eigenes Dateisystem geschaffen; die meisten heute gebräuchlichen Implementierungen nehmen jedoch eine Abbildung auf das Betriebssystem vor, für das sie geschrieben wurden. Man erkennt hier noch sehr schön, welche Funktion SMALLTALK ursprünglich zuge- dacht war: die der einzigen Software auf einem Computer.

Streams auf Dateien

Besonders in SQUEAK gibt es noch zahllose weitere Streams, so u. a. für Multimedia-Aufgaben; insgesamt unterscheiden sich die verschiedenen SMALLTALK-Dialekte bei der Handhabung von Streams zum Teil erheblich, weswegen wir hier auch nicht weiter darauf eingehen.

16 Parallelität: aktive und passive Objekte

Die objektorientierte Weltsicht, die auch in diesem Kurs propagiert wird (nämlich die von den Objekten, die einander Nachrichten schicken und die auf den Empfang von Nachrichten reagieren, indem sie ihren Zustand ändern und weitere Nachrichten verschicken), legt nahe, dass Objekte aktiv sind, will sagen, dass sie über einen eigenen Rechenprozess verfügen. Doch schon in Abschnitt 4.3.2 wurde klar, dass es damit in der Realität nicht weit her ist: Es werden in der Praxis keine Nachrichten verschickt, sondern lediglich Methoden aufgerufen. Abgesehen vom dynamischen Binden dieser Methoden unterscheidet sich damit das Ausführungsmodell der objektorientierten Programmierung nicht von dem der prozeduralen Programmierung (à la PASCAL); insbesondere sind alle Objekte **passiv** (was soviel bedeutet, wie dass sie nur aktiv sind, solange sie gerade eine Methode ausführen).

Unter **aktiven Objekten** würde man sich vorstellen, dass sie über einen Prozess verfügen, der nur die eigenen Methoden ausführt. Erhält ein aktives Objekt eine Nachricht, dann nimmt es diese an und arbeitet sie ab, sobald es die Zeit dazu hat. Die Kommunikation aktiver Objekte würde nämlich asynchron ablaufen, wenn mit der Nachricht (dem Methodenaufruf) nicht auch ein Prozess verbunden ist (was ja dem klassischen Prozederaufruf entspräche). Aktive Objekte wären aber sehr aufwendig und deswegen setzt die objektorientierte Programmierung in der Praxis auf passive.

Gleichwohl ist auch in der objektorientierten Programmierung Parallelverarbeitung möglich. Nur kommt sie (zumindest in SMALLTALK, aber auch z. B. in JAVA) nicht in Form von aktiven Objekten daher, sondern in Form von parallelen Prozessen. Jeder dieser Prozesse führt zu einer Zeit eine Methode aus; er besucht zwar mit dem Methodenaufruf die Empfängerobjekte, diese bleiben jedoch selbst passiv (haben also kein Eigenleben).

parallele Prozesse



Nun gibt es in SMALLTALK eine einfache Möglichkeit, einen neuen Prozess zu starten: Man schickt einfach einem Block die Nachricht **fork**. **fork** entspricht im wesentlichen **value**, nur dass der Block dadurch in einem eigenen, unabhängigen Prozess ausgeführt wird. Entsprechend wartet die Ausführung von **fork** auch nicht darauf, dass die Ausführung des Blocks beendet wurde, bevor sie selbst ein Ergebnis zurückliefert; tatsächlich liefert sie auch nicht (wie **value**) das Ergebnis des Blocks zurück, sondern den Block selbst (als Objekt). Wenn der Block also ein Ergebnis hat, dann geht dieses verloren; aus Sicht des Aufrufers bleiben nur die Seiteneffekte der Ausführung des Blocks, also z. B., wenn sich der Zustand eines der in dem Block vorkommenden Objekte ändert.

Soll ein (paralleler) Prozess nicht sofort starten, so braucht man ein Objekt, das diesen Prozess repräsentiert und dem man dann zu einem späteren Zeitpunkt die Nachricht **resume** schicken kann, die den Prozess startet. Ein solches Objekt erhält man, indem man dem Block **newProcess** schickt. Tatsächlich ist **fork** wie folgt implementiert:

```
701 fork
702   ^ self newProcess resume
```

Um einen parametrisierten Block (also einen Block mit Parametern) als Prozess zu starten, verwendet man statt **newProcess** **newProcessWith:** mit einem Array als Parameter, das die tatsächlichen Parameter des Blocks enthält.

Mit den Nachrichten **suspend** und **terminate** kann man den Prozess dann temporär anhalten bzw. beenden. Angehaltene Prozesse können später mit **resume** wieder gestartet werden, beendete nicht.

Die Synchronisation von parallelen Prozessen erfolgt in SMALLTALK zunächst mittels Semaphoren. Objekte der Klasse **Semaphore** verfügen dazu über zwei Methoden, **wait** und **signal**, und eine Instanzvariable, die für jedes Empfangen von **signal** um 1 erhöht und für jedes Empfangen von **wait** um 1 erniedrigt wird. Wenn ein Prozess **wait** an einen Semaphor schickt, dessen Zähler kleiner 1 ist, wird der betreffende Prozess, der die Nachricht geschickt hat (nicht zu verwechseln mit dem Objekt!) schlafen gelegt (mittels **suspend**). Andernfalls läuft er weiter. Erhält der Semaphor die Nachricht **signal** und es gibt noch Prozesse, die schlafen (erkennbar an einem Zähler kleiner 1), dann kann ein Prozess, der an dem Semaphor wartet, aufgeweckt werden (mittels **resume**) und weitermachen.

Die Synchronisation mittels Semaphoren ist recht elementar und von aktiven Objekten noch weit entfernt. Deutlich näher rückt man mit der Klasse **SharedQueue**, deren Instanzen anstelle von Signalen (die ja einfach nur gezählt werden) Objekte aufnehmen und die eine Synchronisation über **next** und **nextPut:** erlauben. Das Protokoll sieht wie folgt aus:

```
703 next
```



```
704 "Answer the object that was sent through the receiver first and  
705 has not yet been received by anyone. If no object has been sent,  
706 suspend the requesting process until one is."  
  
707 nextPut: value  
708 "Send value through the receiver. If a Process has been suspended  
709 waiting to receive a value through the receiver, allow it to  
710 proceed."
```

Wenn man nun eine solche Shared queue einem Objekt zuordnet und von anderen Objekten verlangt, dass sie Nachrichten, anstatt sie dem Objekt zu schicken (und damit eine Methode des Objekts im eigenen Prozess aufzurufen), in diese Queue einstellen, und dann das Objekt mit einem Prozess, der in einer Endlosschleife läuft, diese Queue auslesen lässt, dann hat man tatsächlich „*aktive Objekte*, die einander Nachrichten schicken“.

17 Lösungen zu den Selbsttestaufgaben

Selbsttestaufgabe 7.1 (Seite 80)

SQUEAK:

```
711 (Smalltalk values) includes: Class ⇒ true
```

SMALLTALK EXPRESS:

```
712 SymbolTable includes: #Class ⇒ true
```

VISUALWORKS:

```
713 (Smalltalk.Core values) includes: Class ⇒ true
```

Alle enthalten nicht nur Klassen.

Selbsttestaufgabe 8.1 (Seite 91)

Weil dann die zu initialisierenden Variablen doch wieder (über den Setter) öffentlich zugänglich sind, was durch **initialize** ja gerade vermieden werden sollte.

Selbsttestaufgabe 8.2 (Seite 91)

```
714 new  
715 |neueInstanz|
```



```

716 AlleInstanzen isNil ifTrue: [AlleInstanzen := Set new].
717 neueInstanz := self basicNew.
718 AlleInstanzen add: neueInstanz.
719 ^ neueInstanz

```

Selbsttestaufgabe 8.3 (Seite 95)

SMALLTALK EXPRESS:

```

720 accessingSubclass: className
721   instanceVariableNamesWithAccessor: instWithAccessor
722   instanceVariableNamesWithoutAccessor: instWithoutAccessor
723   classVariableNames: classVarString
724   poolDictionaries: stringOfPoolNames
725 |newClass|
726 newClass := self
727   subclass: className
728   instanceVariableNames: instWithAccessor, ' ', instWithoutAccessor
729   classVariableNames: classVarString
730   poolDictionaries: stringOfPoolNames.
731 instWithAccessor asArrayOfSubstrings do: [ :aName |
732   newClass compile: (aName , ' ^ ', aName) .
733   newClass compile: (aName, ': argument ', aName, ' := argument. ^ argument') ].
734 ^ newClass

```

Visual Works

```

735 accessingSubclass: className
736   instanceVariableNamesWithAccessor: instWithAccessor
737   instanceVariableNamesWithoutAccessor: instWithoutAccessor
738   classVariableNames: classVarString
739   poolDictionaries: stringOfPoolNames
740 | newClass |
741 newClass := self
742   subclass: className
743   instanceVariableNames: instWithAccessor , ' ' ,
    instWithoutAccessor
744   classVariableNames: classVarString
745   poolDictionaries: stringOfPoolNames.
746 instWithAccessor asArrayOfSubstrings
747 do: [:aName |
748   newClass compile:
      (aName asText) , (' ^ ' asText) , (aName asText) .
749   newClass compile:
      aName , ': argument ' , aName , ' := argument. ^ argument'].
750
751 ^ newClass

```

zusätzlich in der Klasse **String** folgende Methode implementiert:

```

753 asArrayOfSubstrings
754   "Answer an array of substrings from the
755   receiver. The receiver is divided into

```



```

756         substrings at the occurrences of one or
757         more space characters."
758
759 | aStream answer index |
760 answer := OrderedCollection new.
761 aStream := ReadStream on: self.
762 [aStream atEnd]
763   whileFalse: [
764     [aStream atEnd ifTrue: [^ answer asArray].
765      (aStream peek = Character space) not
766      ] whileFalse: [aStream next].
767 index := aStream position + 1.
768 [aStream atEnd or: [aStream peek = Character space]]
769   whileFalse: [aStream next].
770 answer add: (self copyFrom: index to: aStream position)].
771 ^ answer asArray

```

SQUEAK

```

772 accessingSubclass: className
773   instanceVariableNamesWithAccessor: instWithAccessor
774   instanceVariableNamesWithoutAccessor: instWithoutAccessor
775   classVariableNames: classVarString
776   poolDictionaries: stringOfPoolNames
777 | newClass |
778 newClass := self
779 subclass: className
780 instanceVariableNames: instWithAccessor , ' ' ,
781 instWithoutAccessor
782 classVariableNames: classVarString
783 poolDictionaries: stringOfPoolNames
784 category: 'empty'.
785 instWithAccessor subStrings
786 do: [:aName |
787   newClass compile:
788     (aName asText) , (' ^ ' asText) , (aName asText).
789   newClass compile:
790     aName , ': argument ' , aName , ' := argument. ^ argument'].
791 ^ newClass

```

Selbsttestaufgabe 9.1 (Seite 96)

Klassifikation (Umkehrung der Instanziierung): 1., 4., 5., 6., 7.; Generalisierung: 2., 3.

Die längste darin enthaltene Transitivitätskette ist *Elefant ist ein Säugetier ist ein Wirbeltier*. Zwar könnte man meinen, man könne auch noch *Clyde ist ein Elefant* voranstellen (da ja Clyde auch ein Wirbeltier ist), aber dann hätte man bereits zwei verschiedene Interpretationen von Ist-ein gemischt. Jede weitere Anhängung zeigt, was bei einem solchen Mix passieren kann: *Wirbeltier ist ein Stamm* beispielsweise führt zu *Elefant ist ein Stamm* oder gar *Clyde ist ein Stamm*, was offensichtlich Unsinn ist.



Selbsttestaufgabe 10.1 (Seite 108)

791 | c | c := Collection. c new

Selbsttestaufgabe 11.1 (Seite 113)

792 UndefinedObject >> doesNotUnderstand: aMessage
793 ^ self error: 'Null pointer exception'

(<Klassenname >> Methodensignatur ist die in der SMALLTALK-Literatur übliche Schreibweise, um auszudrücken, dass die entsprechende Methode in der genannten Klasse implementiert ist.)

Selbsttestaufgabe 11.2 (Seite 114)

794 (Array with: Object with: Class with: Metaclass) do: [:c1 |
795 (Array with: Object with: Class with: Metaclass) do: [:c2 |
796 Transcript
797 show: c1 printString, ' Subklasse von ', c2 printString,
798 '? ', (c2 allSubclasses includes: c1) printString; cr;
799 show: c1 printString, ' Superklasse von ', c2 printString,
800 '? ', (c2 allSuperclasses includes: c1) printString; cr;
801 show: c1 printString, ' Instanz von ', c2 printString,
802 '? ', (c1 isKindOf: c2) printString; cr; cr]]

Selbsttestaufgabe 12.1 (Seite 117)

Klasse	PrototypicalObject
Superklasse	Object
benannte Instanzvariablen	mDict
Klassenmethoden	

803 new
804 "Answer a new instance of the receiver."
805 ^ super new initialize

Instanzmethoden	
-----------------	--

806 initialize
807 "Private - Initialize the receiver."



```
808     mDict := IdentityDictionary new
809
810     addMethod: methodName withImplementation: aMethod
811     mDict at: methodName put: aMethod
812
813     perform: methodName
814     ^ (mDict
815         at: methodName
816         ifAbsent: [^ super perform: methodName])
817         ) value
818
819     perform: methodName with: firstPara
820     ^ (mDict
821         at: methodName
822         ifAbsent: [^ super perform: methodName with: firstPara])
823         ) value: firstPara
824
825     ...
826
```

Damit die Sprache prototypenbasiert ist, fehlt noch die Angabe eines beerbten Objekts (das die Funktion der Superklasse ersetzt) sowie das automatische Nachschauen in dessen Methodenwörterbuch, falls eine Methode im eigenen nicht gefunden wird.

Selbsttestaufgabe 12.2 (Seite 120)

```
822 Sub new sagMirWerDuBist  ⇒  'ich bin Super'
```

Selbsttestaufgabe 13.1 (Seite 130)

Hier wird vorsintflutlich über die Elemente einer Collection iteriert, die dazu auch noch geordnet sein muss. Stattdessen schreibt man die Iteration natürlich mit **do:!**

Selbsttestaufgabe 13.2 (Seite 130)

Eine entsprechende Methode ist in SQUEAK bereits implementiert, und zwar in der Klasse **Object**:

```
823 caseOf: aBlockAssociationCollection otherwise: aBlock
824     "The elements of aBlockAssociationCollection are associations
825     between blocks. Answer the evaluated value of the first
826     association in aBlockAssociationCollection whose evaluated
827     key equals the receiver. If no match is found, answer the
828     result of evaluating aBlock."
829     aBlockAssociationCollection associationsDo:
```



```
830      [:assoc | (assoc key value = self)
831          ifTrue: [^ assoc value value]].
832      ^ aBlock value
```

Die Wert-Anweisungspaare müssen zuvor in ein Dictionary eingetragen werden.

Selbsttestaufgabe 14.1 (Seite 133)

Es müsste bei jedem Anstoß eines tiefen Kopiervorgangs zunächst eine leere Menge erzeugt werden, in die nach und nach alle kopierten Objekte eingetragen werden. Vor jedem Kopieren müsste dann zunächst geprüft werden, ob das Objekt nicht schon eingetragen, also bereits kopiert worden ist. Falls ja, müsste statt einer neuen Kopie die bereits erzeugte verwendet werden (da sonst identische Objekte nicht zu identischen Kopien führen). Man verwendet für die Buchhaltung am besten ein Dictionary.



Kurseinheit 3: Typen in der objektorientierten Programmierung

The purpose of a type system is to prevent the occurrence of execution errors during the running of a program. The accuracy of this informal statement depends on the rather subtle issue of what constitutes an execution error. Even when that is settled, the type soundness of a programming language (the absence of certain execution errors in all program runs) is a non-trivial property. A fair amount of careful analysis is required to avoid false and embarrassing claims of type soundness; as a consequence, the classification, description, and study of type systems has emerged as a formal discipline.

Luca Cardelli



Im Gegensatz zu SMALLTALK sind die meisten objektorientierten Programmiersprachen typisiert, was soviel heißt wie dass Programmelementen bei ihrer Deklaration (s. Kapitel 19) Typen zugeordnet werden. Dabei schränkt ein Typ die Menge der Objekte, für die ein Programmelement stehen kann, und die Menge der Dinge, die damit gemacht werden können, ein. Meistens sind die Regeln zur Verwendung von Typen fester Bestandteil der Sprache — wenn Sie eine solche Sprache neu lernen, dann würden Sie gar nicht auf die Idee kommen, Typsystem und übrige Sprachdefinition voneinander getrennt zu betrachten. Dennoch sind Typen für das Funktionieren eines Programms prinzipiell verzichtbar⁴⁶ und es lohnt sich durchaus, das Typsystem einer Sprache von ihrem Rest zu lösen, beispielsweise weil man es austauschen oder verbessern will. Dies um so mehr, als heute gängige Typsysteme entweder ziemlich schwach oder ziemlich kompliziert sind.

So führt diese Kurseinheit Typsysteme am Beispiel von STRONGTALK, einer SMALLTALK-Erweiterung um ein *optionales Typsystem*, ein. Sie geht dabei langsam und inkrementell vor. Wer das zu öde erscheint, die sei gewarnt: Es wird noch kompliziert genug und nicht jede Leserin wird alles, was sie in diesem Kurs über Typsysteme liest, auf Anhieb verstehen. Auch wäre

⁴⁶ Wenn man auf Möglichkeiten wie das Überladen von Methoden verzichten kann; Laufzeittypinformation, wie man sie z. B. für das *dynamische Binden* oder für die *Garbage collection* benötigt, kann durch Laufzeitklasseninformation (was nicht dasselbe ist!) ersetzt werden; s. Abschnitt 28.3.



die Alternative, diese Kurseinheit am Beispiel einer bekannteren Sprache mit verpflichtendem Typsystem hochzuziehen, stets mit dem Nachteil belastet, dieses konkrete Typsystem als quasi vom Himmel gefallen darstellen zu müssen — wenn Sie dann später eine andere Sprache kennenlernen, hätten Sie vermutlich Schwierigkeiten, das Gelernte abzustreifen und sich mit den neuen Verhältnissen zurechtzufinden. Ziel dieser Kurseinheit ist aber, dass Sie Typsysteme als das verstehen, was sie sind: eine Möglichkeit zur Spezifikation redundanter Information, die die Qualität von Programmen erhöhen soll.

18 Hintergrund

Sie kennen vielleicht aus anderen Programmiersprachen, dass Variablen und anderen Programmelementen bei ihrer Deklaration (Kapitel 19) ein Typ zugeordnet wird. Dieser Typ schränkt die möglichen Werte der *deklarierten Elemente* ein. So lassen sich beispielsweise in einer Variable vom Typ **Boolean** nur Wahrheitswerte, in einer vom Typ **String** nur Zeichenketten speichern.

Typ ist ein primitiver Begriff, vergleichbar etwa mit dem Begriff der Menge in der Mengentheorie. Ein Typ hat eine *Intension* und eine *Extension*, wobei erstere der Definition des Typs entspricht, letztere seinem Wertebereich, also der Menge der Elemente (Objekte), die zu dem Typ gehören (man sagt auch, „die den Typ haben“ oder „die von dem Typ sind“). Häufig hat ein Typ auch einen Namen, den Typbezeichner. Typen sind die Grundlage von Typsystemen.

Ihnen fällt wahrscheinlich sofort die Ähnlichkeit zum Konstrukt der Klasse, wie es in der letzten Kurseinheit eingeführt wurde, auf. Tatsächlich gibt es hier auch einen Zusammenhang. Um Sie aber nicht gleich in für diese Kurseinheit eher schädliche Denkmuster verfallen zu lassen, soll dieser Zusammenhang zunächst zurückgestellt werden. Eine Aufklärung erfolgt dann in Kapitel 28.

Ein **Typsystem** umfasst Typausdrücke, Objekt- oder Wertausdrücke, Regeln, die Wertausdrücken Typen zuordnen, und Regeln, die von Wertausdrücken einzuhalten sind (zusammen die **Typregeln**). Wertausdrücke (bzw. schlicht Ausdrücke, wenn es nicht um die Abgrenzung von Typausdrücken geht) kennen Sie schon: In SMALLTALK sind es die in Kurseinheit 1, Kapitel 4.1 aufgeführten. Mit den anderen Konzepten werden Sie in den nachfolgenden Kapiteln vertraut gemacht, allerdings in weniger formaler Form, als Sie das nach dieser Definition vielleicht befürchten.

Warum aber typisiert man Variablen und andere Programmelemente? Dafür gibt es mindestens vier gute Gründe:

1. Typisierung regelt das Speicher-Layout.
2. Typisierung erlaubt die effizientere Ausführung eines Programms.

Zusammenhang von
Typ und Klasse

Typsystem



Gründe für die
Typisierung



3. Typisierung erhöht die Lesbarkeit eines Programms.
4. Typisierung ermöglicht das automatische Finden von logischen Fehlern in einem Programm.

Zu 1.: Der Compiler kann anhand des Typs einer Variable bestimmen, wie viel Speicherplatz er für die Aufnahme eines Wertes reservieren muss. Dies ist jedoch naturgemäß nur für Variablen mit Wertsemantik relevant und daher für die objektorientierte Programmierung, insbesondere für Sprachen wie SMALLTALK (in denen Referenzsemantik vorherrscht), von untergeordneter Bedeutung.

Speicher-Layout

Zu 2.: Wenn man weiß, dass die Werte einer Variable immer vom selben Typ sind, also alle demselben Wertebereich entstammen, dann lassen sich bestimmte Optimierungen durchführen. Wenn man z. B. aufgrund der Deklaration einer Variable `x` für gegeben annehmen kann, dass `x` nur ganze Zahlen enthält, dann kann der Compiler für die Übersetzung von `x := x + 1` die Ganzzahladdition, ja sogar die Inkrement-Anweisung des Prozessors verwenden. Kennt der Compiler den Typ von `x` hingegen nicht, dann muss das Programm vor der Ausführung der Addition erst prüfen, von welchem Typ der Wert von `x` ist — handelt es sich um eine Fließkommazahl, so muss es zu der entsprechenden Operation verzweigen, handelt es sich womöglich um gar keine Zahl, dann muss es einen Laufzeitfehler signalisieren oder sich etwas anderes einfallen lassen. Dem kann man entgegenhalten, dass im Falle der objektorientierten Programmierung selbst bei einer Typisierung aller Variablen gelegentlich noch Laufzeittests durchgeführt (oder andernfalls schwere Programmfehler in Kauf genommen) werden müssen, und dass sich die zur Optimierung benötigte Information auch anders als über explizite Typisierung von Variablen (nämlich z. B. über die sog. *Typinferenz*, also die Ausnutzung impliziter Typinformation) gewinnen lässt.

effizientere Ausführung

Zu 3.: In der Vergangenheit hatten Variablen eher kurze, wenig selbsterklärende Namen (vgl. dazu auch Kapitel 62 in Kurseinheit 7). Es ist dann sinnvoll, wenigstens an der Stelle der ersten Erwähnung der Variablen (in der Regel deren *Deklaration*) einen Hinweis darauf zu haben, wofür (für welche Menge von Objekten) die Variable steht. Dies kann über einen Kommentar erfolgen, aber auch durch die Assoziation mit einem Typen, die aussagt, welcher Art die Werte der Variable sein müssen. Doch nicht nur Variablen-, auch Methodennamen können für sich genommen wenig aussagekräftig sein und durch die Verknüpfung mit Typen aussagekräftiger gemacht werden: Eine Deklaration der Methode `next` etwa, die `ListElement` als Typ des Ein- und Ausgabeparameters deklariert, legt nahe, dass sie das in einer Liste auf den Eingabeparameter folgende Element zurück liefert. Ohne die Angabe der Parametertypen müsste man als Nutzerin der Funktion, die ihre Implementation nicht kennt, schon über ihren Zweck spekulieren. Dem mag man freilich entgegenhalten, dass man stattdessen ja auch selbsterklärende Namen für Variablen und Methoden vergeben könnte (mehr dazu in Kurseinheit 7, Kapitel 62).

erhöhte Lesbarkeit



**automatisches
Finden von logischen
Fehlern mittels
Redundanz**

Es bleibt aber in jedem Fall Punkt 4, das Aufdecken von logischen Fehlern in einem Programm. Ohne externes Wissen, was ein Programm tun soll, verlangt das Finden von Fehlern jedoch ein gewisses Maß an **Redundanz**, also die mehrfache Lieferung gleicher Information, im Programm, denn nur wenn eine solche Redundanz vorliegt, können Widersprüche entstehen, die auf einen logischen Programmierfehler hinweisen. Die Verknüpfung von deklarierten Elementen mit Typen erlaubt aber genau die Angabe solcher redundanten Information. Die Schaffung dieser Redundanz verlangt jedoch vermehrte Denk- und Schreibarbeit und ist zudem auch noch, im Falle eines fehlerfreien Programms, überflüssig. Dem kann man allerdings entgegen, dass kaum eine Programmiererin auf Anhieb korrekte Programme schreibt, und wenn eine Typisierung Fehler zu finden in der Lage ist und somit nicht minder aufwendige Tests ersetzt, dann ist das natürlich gut.

Ein fünfter, oben nicht aufgezählter Grund zur Verwendung eines der heute üblichen Typsystems ist übrigens die dadurch entstehende *Modularisierung von Programmen*, nämlich wenn ein Typ zugleich eine *Schnittstelle* oder ein *Interface* ausdrückt. Mehr dazu jedoch erst später (in Abschnitt 28.2).

**Typen als
Schnittstellen**

Die der Fehlerentdeckung mittels Typsystemen zugrundeliegende These ist, dass ein guter Teil logischer Programmierfehler bereits frühzeitig daran erkannt werden kann, dass eine Variable einen Wert hat, den sie eigentlich niemals haben dürfte. So zeugt beispielsweise von einem Fehler, wenn einer Variable, die für Zahlen gedacht war, eine Zeichenkette zugewiesen wird. Wenn dann nämlich einem Ausdruck mit einer arithmetischen Operation, die Zahlen als Operanden verlangt, eine solchermaßen fehlbelegte Variable zugeführt wird, kann dieser nicht ausgewertet werden. Ohne Typprüfung würde dieser Fehler erst zur Laufzeit, also wenn der Ausdruck tatsächlich ausgewertet werden soll, in Erscheinung treten und hätte dann in aller Regel einen Programmabbruch zur Folge. Man nennt einen solchen Programmierfehler einen **Typfehler**.

Typfehler

Während ein Programmabbruch wenigstens noch eine erkennbare Reaktion auf einen Programmierfehler darstellt, ist es fast noch schlimmer, wenn ein logischer Fehler ohne solche bleibt. So kann es beispielsweise vorkommen, dass man einer Variable, deren Inhalt eine Strecke darstellen soll, eine andere zuweist, deren Inhalt eine Zeit repräsentiert. Mit beiden ließe sich gleich rechnen (dieselben Rechenoperationen durchführen), aber das Ergebnis wäre vermutlich falsch. Merken muss man das allerdings selbst, denn das Programm läuft einfach weiter.

**Variablenfehlbele-
gung ohne Typfehler**

Man kann Variablenfehlbelegungen dieser Art verhindern, indem man Variablen mit expliziten **Typinvarianten** versieht, die die Menge ihrer zulässigen Werte beschränken, und dann darüber wacht, dass diese Invarianten immer eingehalten werden. Eine besonders einfache Möglichkeit, solche Invarianten zu spezifizieren, erlauben sog. **Typannotationen**, also die Verbindung einer Variable mit einem Typ, wobei der Typ eine Menge von Werten festlegt, die die Variable ausschließlich haben darf. In typi-

**Typinvarianten und
Typannotationen**



sierter Programmiersprachen erfolgt die Typannotation explizit und zwingend bei der *Variablen-deklaration*; in nicht oder nur optional typisierten Sprachen kann sie auch (für einzelne oder alle Variablen) hergeleitet (inferiert; die *Typinferenz*) werden und ist dann implizit.

Ein Programm, in dem alle Variablenbelegungen immer alle Typinvarianten erfüllen, heißt **typkorrekt**. In einer Sprache, die durch ihr Typsystem

Typkorrektheit; semantische Fehler

Typkorrektheit festzustellen erlaubt, nennt man die logischen Fehler, die sich in unzulässigen Wertzuweisungen ausdrücken, auch **semantische Fehler** (und zwar, weil der Inhalt eines Programmelements nicht seiner intendierten Bedeutung entspricht). Dabei ist die Semantik des Programmelements im Programm zweimal, auf *redundante*, aber unterschiedliche Art, spezifiziert: in Form seines Typs und in Form seiner tatsächlichen Verwendung (festgelegt durch Zuweisungen und Methodenaufrufe). Lässt sich aus beiden ein Widerspruch ableiten, muss eine von beiden falsch gewesen sein.

Der einzige Weg, eine mit der Typisierung einer Variable ausgedrückte Invariante zu verletzen, also Typinkorrektheit herzustellen, ist per Wertzuweisung an die Variable. Ein Typsystem muss also lediglich alle Wertzuweisungen in einem Programm überprüfen, um Freiheit von semantischen Fehlern zu garantieren. Dazu zählen allerdings auch die impliziten Zuweisungen bei Methodenaufrufen (s. Abschnitt 4.3.2), die, auch wegen des dynamischen Bindens, nicht immer alle offensichtlich sind. Im folgenden heißen Zuweisungen und Methodenaufrufe, die nicht zu typkorrekten Programmen führen können, **zulässig**.

Zulässigkeit von Zu- weisungen und Me- thodenaufrufen

Nun kann man sich vorstellen, dass es für einen Compiler selbst in einfachen Fällen nicht leicht ist, festzustellen, ob eine Wertzuweisung eine Invariante verletzt und somit zu einem typinkorrekt Program führt. So ist das folgende STRONGTALK-Programmfragment

Schwierigkeit der Feststellung

```
833 | i <Integer> |
834 |   i := 0.
835 |   i = 0 ifTrue: [i := 1] ifFalse: [i := 'dumm gelaufen']
```

das zunächst eine temporäre Variable **i** mit dem Typ **Integer** (in STRONGTALK wird die *Typannotation* hinter der Variable in spitzen Klammern angeführt) deklariert und ihr dann, in einer Folge von Anweisungen, zunächst **0** und dann **1** (beides Werte vom Typ **Integer**) zuweist, zwar typkorrekt im Sinne obiger Definition, aber um das zu erschließen, muss man schon wissen, dass die Bedingung in Zeile 835 immer erfüllt ist, der False-Zweig, der zu einer Verletzung der Invariante von **i** (nämlich dass die Werte immer vom Typ **Integer** sein müssen und somit nicht vom Typ **String** sein dürfen) führen würde, also nie ausgeführt wird. Im gegebenen Fall ist das zwar offensichtlich (und bereits von einer recht einfachen



Programmanalyse feststellbar), aber es lassen sich auch Fälle konstruieren, in denen eine automatische Programmanalyse streiken muss.⁴⁷

Was man jedoch immer tun kann, um Typkorrektheit zu gewährleisten, ist, dass man zur Laufzeit vor einer Variablenzuweisung prüft, ob der zuweisende Wert den von der Variable geforderten Typ hat. Diese sog. **dynamische Typprüfung** (engl. dynamic type checking) hat jedoch den entscheidenden Nachteil, dass sie zu spät kommt, nämlich zu einem Zeitpunkt, in dem man bereits nicht mehr viel anderes machen kann als einen Fehler zu signalisieren (der dann günstigenfalls durch eine dafür vorgesehene Fehlerbehandlungsmethode aufgefangen wird, der aber in der Praxis häufig nur zu einem Programmabbruch führt). Man kann jedoch argumentieren, dass auch letzteres immer noch besser ist, als mit falschen Werten weiterzuarbeiten und damit entweder einen Programmabbruch an einer anderen Stelle, die nicht mehr so leicht mit der fehlerhaften Wertzuweisung in Zusammenhang zu bringen ist⁴⁸, in Kauf zu nehmen oder gar einen logischen Fehler, der überhaupt nicht erkannt wird.

Man beachte übrigens, dass nach diesem Kriterium SMALLTALK — entgegen häufig zu lesenden Behauptungen — keine dynamische Typprüfung durchführt, da Typfehler erst im letztmöglichen Moment offenbar werden, nämlich wenn auf einer Variable eine Methode aufgerufen werden soll, die für das Objekt, auf das die Variable verweist, gar nicht definiert ist.⁴⁹ Um das zu verhindern, findet man in SMALLTALK-Code gelegentlich Figuren wie

Typprüfung in SMALLTALK

```
836 methodDictionaries: anArray
837     "Private - Change the receiver's array of
838      method dictionaries to anArray."
839     (anArray isKindOf: Array)
840     ifFalse: [^ self error: 'must be an Array'].
841     dictionaryArray := anArray
```

(SMALLTALK EXPRESS entnommen). Dies entspricht natürlich genau einer dynamischen Typprüfung, nur dass hier Typ durch Klasse ersetzt wurde und die Prüfung eben nicht automatisch durch ein Laufzeittypsystem erfolgt, sondern ausprogrammiert werden muss.

⁴⁷ Aus theoretischer Sicht ist das Problem sogar unentscheidbar, auch wenn solche Aussagen in der Regel auf pathologischen Programmkonstruktionen, die man in der Praxis kaum vorfinden wird, basieren.

⁴⁸ Man denke etwa an die sog. *Null pointer exceptions* in JAVA, die erst dann auftreten, wenn mit einem Variablenwert **null** tatsächlich etwas gemacht werden soll, was unter Umständen erst am Ende einer langen Zuweisungskette der Fall ist.

⁴⁹ SMALLTALK und andere Programmiersprachen werden gelegentlich als dynamisch typisiert (dynamically typed) bezeichnet. Das aber ist Unsinn, denn eine Typisierung findet in SMALLTALK gar nicht, auch nicht zur Laufzeit, statt. Außerdem ist mit dynamischer Typisierung in der Regel dynamische Typprüfung gemeint. Was ein dynamischer Typ sein soll, ist auch gar nicht klar.

Sehr viel nützlicher als die dynamische Typprüfung ist die statische Typprüfung, bei der, trotz aller theoretischen Hindernisse, die Typkorrektheit zur Übersetzungszeit gewährleistet werden soll. Die Typprüfung ist damit Aufgabe des Compilers und nicht, wie im Fall der dynamischen Typprüfung, Aufgabe des Laufzeitsystems oder gar der Programmiererin. Wie wir schon gesehen haben, bedeutet dies nicht weniger, als einen Beweis zu führen, dass bei keiner Ausführung eines Programms eine Typinvariante verletzt wird. In der Praxis bedeutet dies aber, dass eine rein statische Typprüfung immer auch Programme zurückweist, die nützlich, sinnvoll und typkorrekt sind (s. obiges Beispiel der Zeilen 833 – 835, das zumindest typkorrekt ist: `i` erhält niemals einen Wert vom Typ `String`). Zwar kann man versuchen, möglichst wenige typkorrekte Programme durch die statische Typprüfung zurückzuweisen, aber wie man sich leicht vorstellen kann, wird mit steigender Genauigkeit das dazu notwendige Typsystem immer aufwendiger und schwieriger zu benutzen, bis es irgendwann so kompliziert ist wie das Programm, dessen Fehler es entdecken soll (so dass man bei auftretenden Typfehlern erst einmal prüfen muss, ob die Ursache tatsächlich in einem fehlerhaften Programm oder vielleicht nur in *fehlerhaften Typannotationen* liegt).

statische Typprüfung

So ist die Suche nach einem guten Typsystem immer die Suche nach einem guten Kompromiss. Die meisten heute in der Praxis verwendeten Typsysteme basieren auf einem solchen: einer statischen Komponente, die möglichst viele Fehler findet, ohne dabei die Programmiererin allzu sehr einzuschränken, und einer dynamischen Komponente, die den Rest erledigt. Eine erwähnenswerte Ausnahme davon macht C++: hier wird, zugunsten von Performanz (Speicherplatz und Geschwindigkeit), auf eine dynamische Komponente der Typprüfung vollständig verzichtet. Da die statische Typprüfung von C++ aber nicht alles abdeckt, sind C++-Programme auch nicht automatisch typkorrekt. Mehr dazu in Abschnitt 51.5.

statisch und
dynamisch in der
Praxis

19 Deklaration, Definition und Verwendung von Programmelementen

Programme bestehen aus Schlüsselwörtern und -zeichen sowie aus Programmelementen, deren Namen, die sogenannten Bezeichner, frei vergeben werden können. Viele Programmiersprachen verlangen, dass man diese Programmelemente vor der ersten Verwendung vereinbart oder deklariert. Durch eine solche **Deklaration** gibt man dem Compiler den Bezeichner bekannt; er kann ihn in der Folge wiedererkennen und mit der Deklaration in Verbindung bringen.

Deklaration

Bei der **Definition** wird dem Bezeichner das zugeordnet, wofür er steht. Im Falle einer Variable ist das eine bestimmte Stelle im Speicher, die genügend Platz bietet, um den Wert der Variable aufzunehmen. Im Falle einer Methode sind es die Anweisungen, die durch die Methode zusammengefasst werden. Nicht selten (aber immer abhängig von der Programmiersprache) erfolgen Deklaration und Definition in einem

Definition
Signatur



Ausdruck. In solchen Fällen spricht man von Deklaration beziehungsweise Definition des Programmelementes in Abhängigkeit davon, was man gerade meint. Bei Variablen ist die Definition in der Regel implizit und aus der Deklaration ableitbar (der Speicherplatz wird vom Compiler automatisch zugewiesen), so dass man hier häufig Deklaration meint, selbst wenn man Definition sagt. Bei Methoden hingegen ist die Unterscheidung essentiell: In ihrer Deklaration wird ihre **Signatur**, das ist ihr Name (in SMALLTALK der Nachrichtenselektor) und die Lister der formalen Parameter, bekanntgegeben, in ihrer Definition wird der Signatur der Methodenrumpf, also die Folge der mit der Methode verbundenen und bei einem Aufruf auszuführenden Anweisungen, zugeordnet. Von der Definition einer Variable zu unterscheiden ist übrigens ihre *Initialisierung*, bei der ihr (der dafür vorgesehenen Speicherstelle) ein Anfangswert zugewiesen wird; in manchen Kontexten (insbesondere im Kontext der Programmanalyse) ist mit Variablendefinition aber auch die Zuweisung eines Werts an eine Variable ganz allgemein gemeint.

Deklaration und Definition dienen letztlich nur einem: der Verwendung. Die **Verwendung** eines Programmelements äußert sich darin, dass sein Name, der Bezeichner, im Programmtext angeführt oder referenziert wird. An der Stelle der Verwendung steht eine Variable für den Wert, den sie hat (bzw., wenn sie auf der linken Seite einer Zuweisung auftaucht, haben soll). Der Bezeichner einer Methode steht hingegen meistens für ihren Aufruf (in manchen Sprachen durch ein Schlüsselwort eingeleitet), seltener auch für einen Zeiger auf die Implementierung.

Deklaration und Definition als Basis für die Verwendung

Variablen-deklarationen haben Sie in SMALLTALK bislang an zwei Stellen gesehen: als *formale Parameter* in *Methodendeklarationen* und als *temporäre, lokale Variablen* in *Methodenrümpfen*. Im Beispiel

Beispiel einer Variablen-deklaration

```
842 methodeM: a mit: b  
843   | c |
```

stecken die Deklarationen von **a** und **b** als formaler Parameter und von **c** als temporäre Variable. Weitere Formen der Deklaration werden Sie im Verlauf dieses Kurstextes noch zu Gesicht bekommen.

In untypisierten Sprachen werden Variablen ohne Angabe eines Typs (wie z. B. in SMALLTALK) oder gar nicht (etliche Skriptsprachen und z. B. BASIC) deklariert. Letzteres hat den erheblichen Nachteil, dass Variablen durch ihre erste Verwendung quasi implizit deklariert (und damit angelegt) werden, was bei Schreibfehlern dazu führt, dass man plötzlich zwei Variablen anstatt einer hat, wobei die eine mit der anderen nichts zu tun hat. Eine solche Einladung zu Programmierfehlern sollten Sie als diejenige, die die Entscheidung für die Auswahl einer Sprache zu treffen hat, stets ablehnen.

fehlende und untypisierte Variablen-deklarationen



20 Typdefinitionen und deren Verwendung

Damit durch ein Typsystem Fehler ausgeschlossen werden können, die auf der Voraussetzung von Eigenschaften von Objekten beruhen, die diese gar nicht haben (also beispielsweise der Verwendung von Nicht-Zahlen in arithmetischen Ausdrücken), muss bekannt sein, welche Eigenschaften einem Typ und damit seinen Elementen zugeordnet sind. Im Fall von SMALLTALK sind die Eigenschaften, die mit einem Objekt verbunden werden können, schnell gefasst: Es handelt sich einfach um die Menge der Methoden, die es versteht, also um sein *Protokoll* (s. Abschnitt 4.3.8 in Kurseinheit 1). Ein solches Protokoll definiert einen Typ: Er umfasst die Menge der Objekte, die über das Protokoll verfügen.

Wenn man nun eine Variable mit einem solchen Protokoll als Typ typisiert und das Programm typkorrekt ist, dann ist garantiert, dass jede Methode, die im Protokoll enthalten ist und die auf der Variable aufgerufen wird, auch für den Inhalt der Variable, das referenzierte Objekt, definiert ist. Typfehler, also Fehler der Sorte „does not understand“ (s. Abschnitt 4.3.2 in Kurseinheit 1), treten dann nicht mehr auf.

Nun kommen in Protokollen aber selbst Variablen vor, nämlich die formalen Parameter der Methoden, die das Protokoll ausmachen. Außerdem ist eine Methode ein Programmelement, das für ein Objekt steht (mit der Ausführung ein Objekt liefert) und deswegen selbst, genau wie Variablen, typisiert werden sollte. Protokolle definieren also nicht nur Typen, sie verwenden auch selbst welche, nämlich indem sie die Typen der Ein- und Ausgabeobjekte spezifizieren. Ein einfaches Beispiel für eine Typdefinition, die selbst Typen verwendet, ist die folgende:

Typen als Teile von
Typdefinitionen

```
844 name ^ <String>
845 name: einName <String> ^ <Self>
846 alter ^ <Integer>
847 alter: einAlter <Integer> ^ <Self>
```

Wie schon bei einer temporären Variable, stehen die *Typannotationen* von formalen Parametern in STRONGTALK in spitzen Klammern dahinter. Diese Schreibweise sollten Sie nicht allzu sehr verinnerlichen, da andere Programmiersprachen die spitzen Klammern zur Kennzeichnung von Typvariablen (in Kapitel 29 behandelt) verwenden. Der Rückgabetyp einer Methode wird durch ein vorangestelltes Dach (^) gekennzeichnet und folgt auf den letzten Parameter. Da es in SMALLTALK keine Methoden gibt, die nichts zurückgeben (eine Methode ohne explizite Rückgabeanweisung gibt in SMALLTALK ja immer das Empfängerobjekt zurück), muss auch immer ein Rückgabetyp angegeben werden. Ist dies der Typ selbst, kann der Name **Self** verwendet werden. Es handelt sich dabei gewissermaßen um eine **Pseudotypvariable** (entsprechend der Pseudovariable **self**, deren Typ sie darstellt).

Falls Sie sich wundern, dass obige Zeilen kein Schlüsselwort zur Einleitung der Typdefinition beinhalten: STRONGTALK ist, genau wie SMALLTALK, ein interaktives, browser-gestütztes System, in dem Typen in Formulare eingetragen und nicht in Textdateien spezifiziert werden. Gleichwohl fällt auf, dass innerhalb der Typdefinition in

Schema für
Typdefinitionen



den spitzen Klammern (also da, wo Typen stehen sollen) keine Typdefinition auftauchen, sondern Namen. Und tatsächlich wird in STRONGTALK jeder Typ benannt (in seiner Typdefinition mit einem Namen versehen). Im folgenden werden Typen, ähnlich wie Klassen, in tabellarischer Form notiert. Der Typ Person etwa mit obigem Protokoll liest sich dann wie folgt:

Typ |

Person

Protokoll |

```
848 name ^ <String>
849 name: einName <String> ^ <Self>
850 alter ^ <Integer>
851 alter: einAlter <Integer> ^ <Self>
```

Selbsttestaufgabe 20.1

Definieren Sie den Typ Boolean gemäß obigem Schema!

In STRONGTALK ist die Protokollbildung der einzige sog. **Typkonstruktor**,

Typkonstruktoren |

d. h., das einzige Sprachkonstrukt, mit dem man neue Typen definieren kann. Andere Programmiersprachen sehen ein reichhaltigeres Angebot vor: In PASCAL beispielsweise gibt es die Typkonstruktoren **record**, **array of**, **set of**, **file of**, Zeiger auf (^) sowie Aufzählungen (enumerations) und Teilbereiche (ranges). In C++ gibt es u. a. **class** und **struct** (entsprechend **record** in PASCAL), JAVA, C# und EIFFEL bieten auch jeweils verschiedene Typkonstruktoren an. Für eine puristische Sprache wie SMALLTALK bzw. STRONGTALK reicht jedoch einer vollkommen aus.

Wie man leicht einsieht, gibt es in STRONGTALK keine primitiven Typen, also keine Typen, deren Definitionen nicht selbst auf einen oder mehrere Typen zurückgeführt werden müsste. Daran röhrt auch die Optionalität der Annotierung nichts: Selbst wenn man eine *Typannotation* weglässt (was immer erlaubt ist), hat die entsprechende Variable bzw. der Rückgabewert der Methode einen Typ, nur wird er an dieser Stelle nicht angegeben. Das wirft natürlich die Frage auf, wie man Typen unter zwangsläufiger Selbstbezüglichkeit überhaupt eine Bedeutung beimesse kann.

20.1 Induktiver Aufbau von Typen und Semantik

Um diese Frage zu beantworten, ist es zunächst interessant, festzustellen, dass es Typen gibt, die sich ausschließlich auf sich selbst beziehen, deren Bedeutung also zumindest nicht von der anderer Typen abhängt. Das klassische Beispiel hierfür ist **Boolean**: Alle seine Operationen fordern den Typ **Boolean** als Operanden und haben **Boolean** als Typ zum Ergebnis. Aber woher erhält **Boolean** seine Bedeutung?





Eine eher theoretisch relevante Möglichkeit, solchen nur auf sich selbst beruhenden Typen eine Bedeutung zu geben, ist, sie auf bekannte externe Formalismen abzubilden. Im Beispiel von **Boolean** ist dies natürlich die *boolesche Algebra*. Jede, die die boolesche Algebra kennt und akzeptiert, wird auch den Typ **Boolean** sofort verstehen und akzeptieren (so er denn den Erwartungen entsprechend definiert ist). Entsprechend lässt sich ein Typ **Fraction** mit den Operationen **+**, **-**, ***** und **/** definieren, der die rationalen Zahlen mit den entsprechenden Operationen repräsentiert. Nimmt man dann noch **Boolean** als mit Bedeutung (Semantik) versehen an, kann man noch Vergleichsoperationen wie **=**, **>**, **<** etc. hinzufügen, ohne in Interpretationsprobleme zu laufen. Andere Typen, für die es eine solche direkte Abbildung nicht gibt, die aber in ihrer Definition rekursiv auf solche Typen zurückgeführt werden können, kann man „induktiv über deren Aufbau“ eine Bedeutung beimessen. Man nennt eine solche Art des Versehens mit Bedeutung eine *denotationale Semantik*.

Eine andere, für die praktische Programmierung relevantere Möglichkeit ist, einen Typ und seine Operationen auf Anweisungen einer (gedachten oder realen, Hauptsache wohlspezifizierten) Maschine abzubilden. Die Abbildung für Basistypen wie **Rational** oder **Boolean** ist in der Programmiersprache bzw. deren Compiler gewissermaßen hart verdrahtet. Für von der Programmiererin definierte Typen kann sie dies hingegen nicht sein; deren Bedeutung kann aber vom Compiler, wiederum „induktiv über deren Aufbau“, aus der Bedeutung von Typen, die eine vorgegebene Semantik haben, abgeleitet werden. Man nennt dies dann auch eine *operationale Semantik*.

Man beachte, dass es für beide Arten der Semantik notwendig ist, dass sich alle Typen auf solche zurückführen lassen, deren Bedeutung vorausgesetzt werden kann. Es gibt also kein vollständig in sich selbst definiertes, von Externem unabhängiges System. Selbst SMALLTALK bzw. STRONGTALK ist kein solches: Auch wenn die Implementierung von **Boolean** nicht „hart verdrahtet“, sondern auf dynamisches Binden abgewälzt wird, so sind dafür aber mindestens die beiden Wahrheitswerte **true** und **false** dem System bekannt, und **Integer** und **Float** (nicht jedoch **Fraction**!) sind „fest verdrahtet“, inklusive der Vergleichsrelationen (die ja die Wahrheitswerte zum Ergebnis haben).

Wenn Sie Kurs 01661 („Datenstrukturen“) bereits belegt haben oder ähnliches Vorwissen besitzen, dann erinnert Sie obiges Schema von Typdefinitionen vielleicht an die Schreibweise abstrakter Datentypen.

Auch dort wird ein Typ syntaktisch als eine Menge von Operationen (Funktionen) beschrieben, deren Operanden (Argumente) alle selbst typisiert sind. Es gibt jedoch mindestens zwei wichtige Unterschiede zwischen den Signaturen eines abstrakten Datentyps und dem Protokoll eines STRONGTALK-Typs:



1. Abstrakte Datentypen sind nicht objektorientiert in dem Sinne, dass die Objekte keinen Zustand haben und bei Operationen (Funktionen) die Objekte, auf denen die Operationen ausgeführt werden, nicht ihren Zustand wechseln. Stattdessen geben Operationen neue Objekte zurück. Die Objekte der abstrakten Datentypen sind also gewissermaßen alle unveränderlich (vgl. Kurseinheit 1, Abschnitt 4.3.5).



- Entsprechend haben die den Methoden eines Protokolls entsprechenden Funktionen in den Spezifikationen abstrakter Datentypen immer ein Argument mehr, und zwar vom Typ des Datentyps selbst. Dieses Argument entspricht in der objektorientierten Programmierung dem Nachrichtenempfänger, dem impliziten Parameter **self**.

Der Bezug zu abstrakten Datentypen ist auch eine beliebte Möglichkeit, Typen einer Programmiersprache mit einer Semantik zu versehen.

20.2 Verwendung definierter Typen

Definierte Typen können in Programmen verwendet werden, in STRONGTALK bei der Deklaration von (anderen) Typen, von Variablen, von Blöcken und von Methoden. Man spricht dann von einer **Typisierung** der deklarierten Programmelemente. Die Verwendung in Typdefinitionen haben Sie ja oben bereits kennengelernt, die Verwendung in Methoden verläuft analog. Variablen (Instanzvariablen, temporäre Variablen etc.) werden in STRONGTALK genau wie formale Parameter (die ja auch Variablen sind) typannotiert, nämlich durch Hintanstellung eines in spitzen Klammern eingeschlossenen Typnamens. Bei Blöcken taucht der Rückgabetyp im selben Segment wie die formalen Parameter auf, also vor dem Separator |. Die vollständig typannotierte Klasse **Stack** aus Abschnitt 8.2 sieht in STRONGTALK beispielsweise so aus:

Klasse	Stack
benannte Instanzvariablen	stackcontent <Array> stackpointer <Integer>
indizierte Instanzvariablen	nein
Instanzmethoden	<pre> 852 push: anElement <Object> ^ <Self> 853 "legt neues Element auf Stapel" 854 stackpointer := stackpointer + 1. 855 stackcontent at: stackpointer put: anElement 856 pop ^ <Self> 857 "entfernt oberstes Element vom Stapel" 858 stackpointer := stackpointer - 1 859 top ^ <Object> 860 "liefert oberstes Element des Stapels" 861 ^ stackcontent at: stackpointer </pre>

Ein Beispiel für einen typisierten Block finden Sie in Abschnitt 29.3, Codezeile 950.



21 Zuweisungskompatibilität

Die Typisierung von Variablen (und anderen Programmelementen — wenn im nachfolgenden nur von Variablen die Rede ist, dann sind letztere meistens mit gemeint) soll also bewirken, dass in einem Programm jede Variable nur die Werte haben kann, für die sie (die Variable) vorgesehen ist (die Einhaltung der *Typinvariante*). Voraussetzung dafür ist zum einen, dass jeder Variable ein Typ zugeordnet ist, zum anderen, dass auch jedes Objekt sowie jeder Ausdruck, der für einen Wert oder ein Objekt steht, einen Typ hat. Ersteres geschieht in sogenannten Variablen-deklarationen, letzteres ergibt sich aus den zu einem *Typsystem* gehörenden *Regeln zur Zuordnung eines Typs zu Ausdrücken*, nämlich

- bei Literalen aus der Art des Literals, dessen Typ dem Compiler bekannt ist,
- bei der Instanziierung aus dem noch zu klärenden Zusammenhang von der instanzierten Klasse mit den Typen eines Programms sowie
- bei Nachrichtenausdrücken aus der Deklaration der dazugehörigen Methode, die ja (genau wie eine Variablen-deklaration) angeben muss, welchen Typs die Objekte sind, die sie liefert.

Es bleibt die Frage nach den ebenfalls zu einem Typsystem gehörenden *Typregeln, die von Ausdrücken einzuhalten sind*, nämlich wie die Typkorrektheit bzw. andernfalls die Verletzung einer Typinvariante genau festgestellt wird. Es ist ja bereits klar, dass es dazu ausreicht, die Wertzuweisungen in einem Programm zu überprüfen. Diese Überprüfung findet in der Regel in Form der Feststellung der sog. *Zuweisungskompatibilität* statt. Die Sprachregelung ist hier leider nicht ganz einheitlich, aber im folgenden gehen wir davon aus, dass alle typisierten Sprachen den Begriff der Zuweisungskompatibilität kennen und sich lediglich in ihren Definitionen der Regeln, die für das Bestehen einer Zuweisungskompatibilität eingehalten werden müssen, unterscheiden. Vor allem darum wird es in den nächsten Kapiteln gehen.

Feststellung der
**Zuweisungskompati-
bilität**

Angenommen, zwei temporäre Variablen `anzahl` und `erfolgreich` seien wie folgt deklariert:

862 | `anzahl <Integer> erfolgreich <Boolean>` |

Dann sind, unter der Annahme, dass `12` vom Typ `Integer` ist und `true` vom Typ `Boolean`, die Zuweisungen

863 `anzahl := 12`
864 `erfolgreich := true`

zulässig (da sie keine Typinvariante verletzen),

865 `anzahl := false`
866 `erfolgreich := 12`



hingegen nicht. Ist eine Zuweisung zulässig, dann spricht man auch von einer **Zuweisungskompatibilität** der beteiligten Typen. Die für das Programmieren relevante Implikation ist allerdings die umgekehrte: Wenn zwei Typen zuweisungskompatibel sind, dann gilt, dass eine entsprechende Zuweisung *zulässig* ist, also zu keiner Verletzung einer Typinvariante führt. Wie Sie noch sehen werden, verlangt Zuweisungskompatibilität keineswegs identische Typen; daraus ergibt sich aber eine sprachliche Uneindeutigkeit, die zunächst behoben werden muss.

Dem Satz „**a** ist zuweisungskompatibel mit **b**“ kann man nicht eindeutig entnehmen, ob nun **a** **b** zugewiesen werden kann oder **b** **a**. Dass beides geht, ist nur dann der Fall, wenn die beteiligten Typen äquivalent in einem noch zu bestimmenden Sinne sind, was aber, wie schon gesagt, nicht unbedingt der Fall sein muss. Im folgenden soll daher die Richtung der erlaubten Zuweisung so gelesen werden, dass beim Satz „**a** ist zuweisungskompatibel mit **b**“ die Zuweisung **b** := **a** zulässig ist. Die umgekehrte Richtung, **a** := **b**, kann ebenfalls zulässig sein; dies wird durch den Satz jedoch nicht ausgesagt. Zuweisungskompatibilität ist übrigens (in der Regel) eine transitive Eigenschaft: Wenn **a** zuweisungskompatibel mit **b** ist und **b** zuweisungskompatibel mit **c**, dann ist auch **a** zuweisungskompatibel mit **c**.

Sprachgebrauch; Transitivität

Auch bei impliziten Zuweisungen wie der Parameterübergabe von Methodenaufrufen (den dabei stattfindenden Zuweisungen der tatsächlichen an die formalen Parameter; s. Abschnitt 4.3.2) impliziert Zuweisungskompatibilität Typkorrektheit. Außerdem kann eine Methode, wenn sie Werte zurückgibt, ja selbst in rechten Seiten von Zuweisungen auftreten; der Typ dieser Werte muss dann mit der Variable auf der linken Seite zuweisungskompatibel sein. So sind bei Vorliegen der Deklarationen

Zuweisungskompatibilität bei Methodenaufrufen

```
867 m: p <E> ^ <A>
868 | e <E> a <A> |
```

sowohl die explizite als auch die impliziten Zuweisungen in

```
869 a := self m: e
```

zulässig; den Methodenaufruf kann man im übertragenen Sinne als *zulässig* bezeichnen.

22 Typäquivalenz

Es stellt sich nun die Frage, wann ein Typ mit einem anderen zuweisungskompatibel ist. Offensichtlich ist dies der Fall, wenn die Typen dieselben (identisch) sind. Wie bereits oben erwähnt, ist dies aber keine notwendige Voraussetzung für die Zuweisungskompatibilität. Es ist nämlich zumindest auch möglich, dass sich zwei verschiedene Typdefinitionen bis auf ihre Namen gleichen, dass also z. B. in STRONGTALK die die Typdefinitionen ausmachenden



Mengen der Methodensignaturen gleich sind. Man spricht in diesen Fällen von einer **Typäquivalenz**.

Von der Typäquivalenz gibt es zwei Arten: die **nominale** (sich auf den Namen beziehende) Typäquivalenz, auch **Namensäquivalenz** genannt, und die **strukturelle** Typäquivalenz, auch als **Strukturäquivalenz** bezeichnet. Während die nominale Typäquivalenz verlangt, dass zwei Deklarationen (beispielsweise von Variablen) dieselben Typen anführen, damit Zuweisungskompatibilität vorliegt, kommt es bei der strukturellen Typäquivalenz lediglich darauf an, dass die Typen paarweise gleich definiert sind (also die gleichen Eigenschaften von ihren Werten verlangen), die Typen sich also in ihrer Struktur, aber nicht unbedingt in ihren Namen gleichen.

Unterscheidung von nominaler und struktureller Typäquivalenz

Typäquivalenz ist eine symmetrische Eigenschaft: Wenn ein Typ A (nominal oder strukturell) äquivalent zu einem Typ B ist, dann ist B genauso äquivalent zu A. Die Reflexivität der Typäquivalenz, also dass jeder Typ äquivalent zu sich selbst ist, ergibt sich von selbst. Außerdem ist Typäquivalenz transitiv: Wenn A (nominal oder strukturell) äquivalent zu B ist und B in der gleichen Art äquivalent zu C, dann ist auch A äquivalent zu C (und, aufgrund der Symmetrie, C äquivalent zu A).

formale Eigenschaften der Typäquivalenz

22.1 Strukturäquivalenz

Um strukturelle Typäquivalenz festzustellen, werden die Definitionen der beteiligten Typen *rekursiv expandiert*, was soviel heißt wie dass in einer Typdefinition vorkommende Namen anderer Typen durch ihre Struktur ersetzt werden. Nimmt man beispielsweise die Typdefinitionen

Typ | Person

Protokoll |

```
870  sitz ^ <Wohnung>
871  sitz: einWohnsitz <Wohnung> ^ <Self>
```

Typ | Wohnung

Protokoll |

```
872  straße ^ <String>
873  straße: einStraße <String> ^ <Self>
874  ort ^ <String>
875  ort: einOrt <String> ^ <Self>
```

Typ | Firma

Protokoll |

```
876  sitz ^ <Büro>
877  sitz: einFirmensitz <Büro> ^ <Self>
```

Typ | Büro



```
878 straße ^ <String>
879 straße: einStraße <String> ^ <Self>
880 ort ^ <String>
881 ort: einOrt <String> ^ <Self>
```

dann sind die Typen **Person** und **Firma** sowie **Wohnung** und **Büro** jeweils strukturäquivalent, aber nicht namensäquivalent. Bei der Strukturäquivalenz haben Namen also lediglich die Funktion der abkürzenden Schreibweise, bei der Namensäquivalenz hingegen auch eine von der Struktur unabhängige Bedeutung. Namensäquivalenz impliziert Strukturäquivalenz, aber nicht umgekehrt; Namensäquivalenz ist somit das stärkere Konzept.

Strukturäquivalenz als Bedingung der Zuweisungskompatibilität reicht aus, um *Typfehler*, also logische und Laufzeitfehler, die auf der Annahme einer nicht vorliegenden Eigenschaft (Methode) bei einem Wert einer Variable basieren, zu verhindern. Sie garantiert, dass die Methoden eines Programms auf den jeweiligen Empfängerobjekten mit den geforderten Parameterobjekten auch durchgeführt werden können. So kann z. B. bei erfolgreicher Typprüfung (und daher vorliegender Typkorrektheit) ohne Kenntnis der konkreten Inhalte der Variablen sichergestellt werden, dass bei Vorliegen der Deklaration `p <Person>` der Ausdruck

Bedeutung der Strukturäquivalenz

```
882 p sitz straße: 'Heimatstraße'
```

keine Typfehler produziert, und gleichzeitig der Ausdruck

```
883 p sitz: 'zuhause'
```

schon zur Übersetzungszeit als fehlerhaft zurückgewiesen wird, da er zu einer Variablenfehlbelegung (die in SMALLTALK noch problemlos möglich gewesen wäre) führt. Man beachte, das letztere sogar zu einer Speicherschutzverletzung führen könnte, wenn die Variable `p` — wie in vielen Sprachen mit Typsystem — Wertsemantik hätte, nämlich dann, wenn der übergebene String größer ist als der zur Aufnahme der Wohnung vorgesehene Speicherplatz.

Strukturäquivalenz ist eine rein syntaktische Bedingung. Insbesondere können bei geforderter Strukturäquivalenz Typen zufällig zuweisungskompatibel sein, die inhaltlich überhaupt nichts miteinander zu tun haben. Dadurch können Objekte, die eigentlich getrennten Typen (disjunkten Wertebereichen) angehören, über Kreuz und über die Typgrenzen hinweg zugewiesen werden. *Semantische Fehler* sind also immer noch möglich. Man trifft daher in Sprachen mit Strukturäquivalenz gelegentlich die Praxis an, jedem Typ eine für ihn charakteristische Methode exklusiv zuzuordnen, so dass er mit keinem anderen mehr strukturäquivalent ist. Diese Technik nennt man **Type branding**.

Type branding



22.2 Namensäquivalenz

Nun können Typen neben ihrer formalen Funktion, Fehler zu vermeiden, noch eine inhaltliche, nämlich eine *Filterfunktion* ausfüllen. Diese setzt allerdings voraus, dass dem Typ auch eine Bedeutung, die über seine bloße Struktur (seine Syntax) hinausgeht, beigemessen werden kann. Dies geschieht heute vor allem durch die Benennung des Typs, die dann, gepaart mit Namensäquivalenz als Bedingung der Zuweisungskompatibilität, verlangt, dass einer Variable nur Werte gleicher Bedeutung zugewiesen werden können. Eine Zuweisung einer Wohnung an ein Büro oder umgekehrt ist dann, trotz im obigen Beispiel strukturell gleich definierter Typen und deswegen ausbleibenden Typfehlern, aufgrund fehlender Namensgleichheit ausgeschlossen, was auch sinnvoll ist, da es sich dabei mit einer gewissen Wahrscheinlichkeit um einen logischen Programmierfehler handelt, der auf mechanische Art sonst kaum zu entdecken wäre. Die Filterfunktion der geforderten Namensäquivalenz drückt also eher eine Absicht der Programmiererin aus denn eine technische Notwendigkeit. Die Bedeutung gerade dieser Funktion sollte man jedoch nicht unterschätzen — nur wenige Möglichkeiten, Fehler in einem Programm aufzudecken bzw. zu vermeiden, sind so einfach zu haben.

inhaltliche
Filterfunktion der
Typprüfung

Ein der Typprüfung per Namensäquivalenz ähnliches Prinzip kommt übrigens in der Physik zur Anwendung: Bei ihren Berechnungen führen Physikerinnen stets eine Art Typprüfung durch, indem sie nicht nur mit den Beträgen der physikalischen Größen, sondern auch mit deren Einheiten rechnen. Wenn Physikerinnen also beispielsweise eine Geschwindigkeit berechnen und bei der Behandlung der Einheiten etwas anderes als m/s herauskommt, dann steckt im Rechenvorgang ein Fehler — das Ergebnis hat nicht den richtigen Typ (die richtige Einheit) und ist deswegen mit hoher Wahrscheinlichkeit falsch.

Rechnen mit Größen

Namensäquivalenz hat aber auch einen entscheidenden Nachteil: Sie setzt voraus, dass getrennt voneinander entwickelte Programme zumindest an ihren Schnittstellen (also da, wo Objekte ausgetauscht werden) dieselben Typen verwenden. Dies kann für die Interoperabilität von getrennt voneinander entwickelten Programmen (wie z. B. Web services) ein echtes Hindernis sein.

Nachteil der
Namensäquivalenz

Strukturelle Typäquivalenz bietet mehr Flexibilität als nominale: Sie erlaubt Äquivalenz von Typen, bei deren Definition man vom jeweils anderen nichts wusste. Die erhöhte Flexibilität hat jedoch ihren Preis: Zufällige strukturelle Übereinstimmungen können zu einer Äquivalenz führen, die nicht der intendierten Semantik entspricht. *Type branding* führt in solchen Fällen eine Namensäquivalenz durch die Hintertür ein, mit dem Vorteil, dass diese optional ist.

strukturell vs.
nominal



23 Typerweiterung

Wie bereits in Kapitel 21 angedeutet, verlangt die Zuweisungskompatibilität nicht unbedingt Typäquivalenz. Tatsächlich reicht es ja, bei einer rein strukturellen (syntaktischen) Betrachtung, voll aus, dass der Typ der rechten Seite einer Zuweisung das Protokoll (die Menge der Methoden) des Typs der linken Seite enthält, um in der Folge Typfehler zu vermeiden. Anders ausgedrückt: Der Typ auf der rechten Seite einer Zuweisung darf eine Erweiterung dessen auf der linken Seite um zusätzliche Methoden sein.

Die sog. **Typerweiterung** (engl. type extension; extension hier im Sinne von Erweiterung und nicht im Sinne der Ausdehnung als Gegenstück zur Intension; vgl. Abschnitt 7.1 in Kurseinheit 2), wie sie z. B. in den Programmiersprachen MODULA-3 und OBERON (beides Nachfolger von PASCAL) Verwendung findet, sieht genau dies vor. Eine Typerweiterung des obigen Typs **Büro** um ein Länderkennzeichen sieht dann beispielsweise wie folgt aus:

Vererbung der Methodendeklaration



Typ	InternationalesBüro
erweiterter Typ	Büro
Protokoll	
884	länderkennzeichen ^ <String>
885	länderkennzeichen: einLänderkennzeichen <String> ^ <Self>

Der erweiternde Typ, hier **InternationalesBüro**, wird also relativ zu einem bereits bestehenden, dem erweiterten Typ (hier **Büro**), definiert. Die Methodendeklarationen des erweiterten Typs werden dabei gewissermaßen an den erweiternden Typ *vererbt*; dieser braucht sie also nicht noch einmal zu wiederholen.

Wie man nun leicht einsieht, können Variablen, deren deklarerter Typ **Büro** ist, auch Objekte vom Typ **InternationalesBüro** enthalten, ohne dass dies zu Typfehlern führt, da alle Methoden, die für **Büro** vorgesehen sind, auch in **InternationalesBüro** vorkommen.⁵⁰ Das Umgekehrte ist jedoch nicht der Fall: Wenn man einer Variable vom Typ **InternationalesBüro** ein Objekt vom Typ **Büro** zuweisen könnte, dann hätte man immer dann ein Problem, wenn man über diese Variable auf dessen Methoden zu Länderkennzeichen zugreifen wollte, weil diese

Zuweisungskompatibilität bei Typerweiterung

⁵⁰ Bei Variablen mit Referenzsemantik geht das ohne Einschränkungen, denn die Größe des durch einen Pointer belegten Speicherplatzes ist immer gleich. Bei Variablen mit Wertsemantik hingegen muss der Wert eines erweiterten Typen erst auf einen des Basistypen projiziert werden, d. h., die Inhalte eventueller zusätzlicher Felder müssen unter den Tisch fallen, da für sie im für die Variable reservierten Speicher kein Platz ist. Solange in den Typdefinitionen aber gar keine Felder vorkommen, ist der Typ einer Variable auch nicht für die Berechnung des zur Aufnahme eines Objekts des Typs benötigten Speichers geeignet. Das ist z. B. in STRONGTALK der Fall — und auch gut so, denn Felder zählen nach vorherrschender Meinung zur Implementation und sind, genau wie bei abstrakten Datentypen, nicht Bestandteil einer Typdefinition.



schlichtweg für das Objekt nicht definiert sind. Die Zuweisungskompatibilität unter Typweiterung regelt der Begriff der Typkonformität.

24 Typkonformität

Einen Typ, dessen Definition alle deklarierten Elemente der Definition eines anderen Typen enthält, nennt man mit dem anderen **typkonform**. So ist **InternationalesBüro** im obigen Beispiel mit **Büro** typkonform. Typkonformität ist in vielen Sprachen eine notwendige und hinreichende Voraussetzung für die *Zuweisungskompatibilität*: Es darf dann ein Objekt vom Typ **InternationalesBüro** einer Variable vom Typ **Büro** zugewiesen werden.

Typkonformität ist aber reflexiv, d. h., jeder Typ ist konform zu sich selbst. Sie ist weiterhin transitiv: Wenn A typkonform zu B ist und B typkonform zu C, dann ist auch A typkonform zu C. Wie man sich leicht denken kann, ist die Typkonformität jedoch im Gegensatz zur Typäquivalenz nicht symmetrisch: Aus der Tatsache, dass ein Typ B typkonform zu einem Typ A ist, folgt nicht, dass auch A typkonform zu B ist. Vielmehr ist dies mit einer kleinen Ausnahme sogar zwingend nicht der Fall: Typkonformität ist meistens antisymmetrisch, was soviel heißt wie dass wenn B zu A und A zu B typkonform ist, dass dann A und B identisch sein müssen.

**formale
Eigenschaften der
Typkonformität**

Von der Typkonformität gibt es, genau wie von der Typäquivalenz, zwei Varianten, nämlich eine **strukturelle Typkonformität** und eine namensgebundene (**nominale**) **Typkonformität**. Zur strukturellen Typkonformität reicht es aus, wenn der konforme Typ wie oben alle Elemente des Typs, zu dem er konform sein soll, enthält: Der Typ mit der Definition

**strukturelle und
nominale
Typkonformität**

Typ	InternationaleWohnung
Protokoll	
886	straße ^ <String>
887	straße: einStraße <String> ^ <Self>
888	ort ^ <String>
889	ort: einOrt <String> ^ <Self>
890	länderkennzeichen ^ <String>
891	länderkennzeichen: einLänderkennzeichen <String> ^ <Self>

ist also zum Typ **Büro** strukturell konform. Für die nominale Konformität muss zusätzlich und explizit die Erweiterung eines (oder Ableitung von einem) anderen Typ angegeben werden: die Definition von **InternationalesBüro** aus Kapitel 23 ist also mit **Büro** nicht nur strukturell, sondern auch nominal konform. Da bei der Erweiterung alle Elemente des Typs, der erweitert wird, beim erweiternden erhalten bleiben, folgt die Konformität aus der Erweiterung.



Nun ist die Teilmengenbeziehung reflexiv, was auf die Typerweiterung übertragen bedeutet, dass ein Typ eine Erweiterung eines anderen sein kann, ohne tatsächlich etwas hinzuzufügen. So ist beispielsweise gemäß folgender Typdefinition

Typ	NationalesBüro
erweiterter Typ	Büro
Protokoll	

NationalesBüro eine Erweiterung von **Büro** und mit den Variablendeklärationen

892 | b <Büro> nb <NationalesBüro> |

die Zuweisung

893 b := nb

bei geforderter nominaler und struktureller Typkonformität zulässig. Die umgekehrte Zuweisung ist

894 nb := b

ist jedoch bei geforderter nominaler Typkonformität nicht zulässig, da **Büro** eben *nicht* nominal konform ist zu **NationalesBüro**; strukturell ist es es hingegen schon.

Typäquivalenz impliziert übrigens, jeweils für die nominale und die strukturelle Form getrennt, Typkonformität: Zwei äquivalente Typen sind auch immer konform. Das Umgekehrte ist jedoch meistens nicht der Fall: Zwar ist ein Typ, der angibt, einen anderen zu erweitern, ohne jedoch etwas hinzuzufügen, zu dem anderen strukturell äquivalent, aber nominal schon nicht mehr; sobald etwa hinzugefügt wird, ist es mit der Äquivalenz sowieso vorbei.

**Typäquivalenz als
Spezialfall der
Typkonformität**

Genau wie bei der Typäquivalenz hat die nominale Typkonformität zusätzlich zur Gewährleistung der Zuweisungskompatibilität und somit der Abwesenheit von Typfehlern (die ja auch bei einer strukturellen Typkonformität schon gegeben wäre) eine *Filterfunktion*: Es sind nur Objekte von solchen Typen Variablen zuweisbar, für die das die Programmiererin aufgrund semantischer (inhaltlicher) Überlegungen ausdrücklich so vorgesehen hat. Auf diese Filterfunktion werden wir später im Zusammenhang mit sog. *Tagging* oder *Marker interfaces* (in Kurseinheit 4, Kapitel 45) noch zurückkommen.

Filterfunktion

Da die Typkonformität bei Nennung des Typen, von dem ein neuer per Erweiterung abgeleitet wird, über den Vorgang der Erweiterung automatisch gegeben ist (und so keine aufwendigen, fallweisen Konformitäts-tests durchgeführt werden müssen), setzen die meisten gebräuchlichen, typisierten Programmiersprachen auf nominale Typkonformität als Bedingung für die Zuweisungskompa-

**Vorteile der
nominalen
Typkonformität**

tibilität. Interessanterweise wurde STRONGTALK, das ursprünglich ein auf struktureller Konformität beruhendes Typsystem (inkl. *Type branding*) hatte, inzwischen auf nominale Typkonformität umgestellt. Als Begründung wurde angeführt, dass ein strukturelles Typsystem, insbesondere eines, bei dem Typen nicht explizit benannt werden, es der Programmiererin nicht erlaubt, ihre Absicht (intendierte Semantik, die obengenannte Filterfunktion) auszudrücken, was Programme schwerer zu lesen und zu debuggen macht, und dass die Fehlermeldungen, die eine strukturelle Typprüfung produziert, sich oft nicht auf die eigentliche Fehlerquelle beziehen und sehr schwer zu verstehen sind [STRONGTALK 2.0].

Fragen der Zuweisungskompatibilität unter Typerweiterung spielen übrigens auch bei Funktionsaufrufen, bei denen ja *implizite Zuweisungen* auftreten (s. Abschnitt 4.3.2), eine wichtige Rolle. So muss bei dem Ausdruck

Konformität bei Funktionsaufrufen

895 `a := self m: e`

der Typ von `e` eine Erweiterung des in `m` für den Parameter geforderten Typ sein und der Rückgabetyp von `m` eine Erweiterung des Typs von `a`.

25 Typeinschränkung

Typerweiterung ist nicht die einzige Möglichkeit, auf der Basis eines bereits bestehenden einen neuen, verwandten Typen zu erzeugen; Typeinschränkung ist eine andere.

Eine erste, offensichtliche Form der Typeinschränkung liegt dann vor, wenn ein Typ auf Basis eines anderen unter Entfernen von Eigenschaften (Methoden) definiert wird (das Beispiel vom Pinguin als einem Vogel, der nicht fliegen kann, kennen Sie ja bereits aus Kurseinheit 1, Abschnitt 9.2; das Beispiel vom Quadrat als einem Rechteck, das nur eine Kantenlänge braucht, ist ein anderes). Diese Form der Typeinschränkung stellt zumindest auf Ebene der Typdefinition (der Intensionen) die Umkehrung der Typerweiterung dar. Es liegt auf der Hand, dass diese Form der Typeinschränkung nicht zur Zuweisungskompatibilität führt; dies folgt schon aus der fehlenden Symmetrie der Typkonformität. Sie soll hier deswegen keine weitere Berücksichtigung finden, auch wenn es Sprachen gibt, die sie erlauben (z. B. EIFFEL).

Typeinschränkung durch Lösen von Methoden

Eine unter dem Gesichtspunkt der Zuweisungskompatibilität interessantere Form der Typeinschränkung besteht darin, die verwendeten Typen einer Typdefinition durch andere, speziellere zu ersetzen (ohne hier schon zu sagen, was „spezieller“ im Zusammenhang mit Typen bedeutet). Diese Form der Typeinschränkung ergibt sich auf natürliche Weise, wenn man sich den Zusammenhang von Extensionen von definierten Typen und solchen, die in Typdefinitionen vorkommen, ansieht.

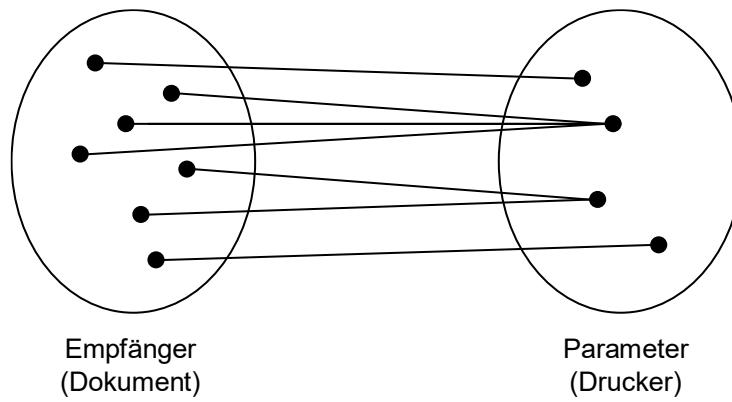
Typeinschränkung durch Spezialisierung

Das Ganze soll an einem Beispiel verdeutlicht werden. Man denke sich einen Typ **Dokument** wie folgt definiert:



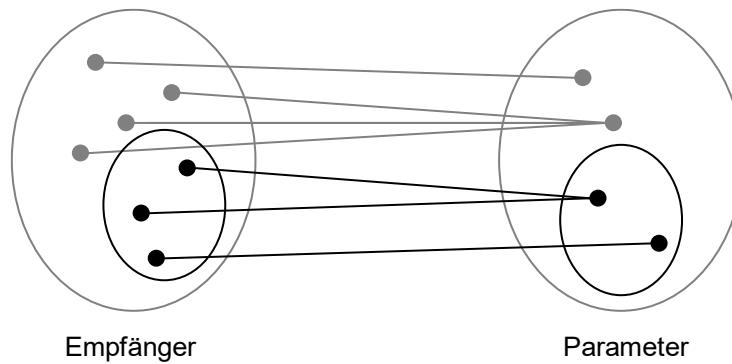
```
896 name ^ <String>
897 name: einString <String> ^ <Self>
898 druckenAuf: einemDrucker <Drucker> ^ <Self>
```

Für die Dauer eines Ausdrucks wird durch die Methode `druckenAuf`: ein konkretes Dokument einem konkreten Drucker zugeordnet. Mengentheoretisch betrachtet ist diese Zuordnung eine Relation zwischen zwei Mengen:



Nun gibt es verschiedene Arten von Dokumenten und Druckern: Man kann z. B. bei Dokumenten zwischen Texten und Diagrammen unterscheiden und bei Druckern zwischen Zeilendruckern und Plottern. Die Extensionen entsprechender Typen sind dann jeweils Teilmengen der Extensionen von **Dokument** und **Drucker**.

Weiterhin ergibt es sich aus der Natur der Sache, dass man Diagramme nur auf Plottern drucken sollte und Texte nur auf Zeilendruckern. Dies geht konform zur obigen Betrachtung eines Methodenauftrags als Relation: Wenn man die Menge einer Stelle einer Relation wie der obigen auf eine Teilmenge einschränkt, dann schränkt sich dadurch in der Regel auch die Menge der anderen Stelle auf eine Teilmenge ein:



Die andere Menge kann auch gleichbleiben; größer wird sie jedoch nie.

Es ergibt sich daraus die folgende Definition eines Typs **Zeichnung** als Typeinschränkung von **Dokument**:



Typ

Zeichnung

eingeschränkter Typ

Dokument

Protokoll

899 druckenAuf : einemDrucker <Plotter> ^ <Selbst>

Man beachte, dass die Methode **druckenAuf** : nicht hinzugefügt wurde

Redefinition

— sie ersetzt vielmehr die von **Dokument** übernommene. So unterscheidet sich die Methode von der ursprünglichen auch nur in der *Typeannotation* des formalen Parameters. Man spricht in diesem Zusammenhang von einer **Redefinition** der Methode (an den ebenfalls dafür verwendeten Begriff des *Überschreibens* sind je nach Programmiersprache andere Bedingungen geknüpft). Die Methoden **name** und **name** : werden übrigens, genau wie bei der Typerweiterung, bei der Typeinschränkung übernommen, solange nichts anderes ausgesagt wird.

Man mag sich fragen, warum bei der Typerweiterung in Kapitel 23 keine zwei Formen analog zur Typeinschränkung eingeführt wurden. Die Typerweiterung würde damit zur vollständigen Umkehrung der Typeinschränkung wie hier beschrieben. Wie Sie noch sehen werden, ist das Ziel nicht die Schaffung zweier Komplementäre, sondern die Vereinigung beider zu einer Beziehung zwischen Typen — dazu müssen sie aber in dieselbe und nicht in gegensätzliche Richtungen gehen. Außerdem ist eine Erweiterung des Wertebereichs bei Einschränkung des Definitionsbereichs nicht durch den Begriff der Relation wie oben erklärbar; eine wichtige Analogie zur Realität, die durch Typen zwecks semantischer Prüfung nachgebildet werden soll, ginge damit verloren.

**Verhältnis von
Typeinschränkung
und Typerweiterung**

Nun ergibt sich aber bei der Typeinschränkung auch ohne Löschen das Problem, dass sie die Zuweisungskompatibilität, die ja für die Typerweiterung noch per Typkonformität geregelt werden konnte, aushebelt: Wenn man bei den obigen Typdefinitionen und den Deklarationen

**Typeinschränkung
und Zuweisungs-
kompatibilität**

900 | d <Dokument> z <Zeichnung> l <Zeilendrucker> |

zunächst

901 d := z

zuweist und dann weiter

902 d druckenAuf : l

aufruft, dann wäre, Typkonformität von **Zeichnung** und **Dokument** bzw. **Zeilendrucker** und **Drucker** vorausgesetzt, die Typprüfung zwar erfolgreich (denn **druckenAuf** : verlangt für **Dokument Drucker** als Argumenttyp), aber zur Laufzeit soll nun eine Zeichnung auf einem Zeilendrucker gedruckt werden, was gemäß der obigen Definition von **Zeichnung** nicht vorgesehen ist. Eine der beiden Zuweisungen, die explizite in



Zeile 901 oder die implizite (die Parameterübergabe) in Zeile 902, ist also nicht zulässig. Da gegen die Typkonformität von **Zeilendrucker** und **Drucker** nichts spricht (für beide Typen sind im Beispiel ja gar keine Definitionen angegeben), bleibt nur, dass **Zeichnung** nicht typkonform zu **Dokument** ist, wobei der Grund hierfür in der Einschränkung des Parametertyps von **druckenAuf**: bei der Redefinition zu suchen ist.

Wesentlich für diese Betrachtungsweise, und damit das geschilderte Problem, ist übrigens, dass nach der Zuweisung von Zeile 901 **d** und **z** auf dasselbe Objekt, nämlich eine **Zeichnung**, verweisen. **d** ist also ein *Alias* für **z** (s. Kurseinheit 1, Abschnitt 1.8). Unter *Wertsemantik*, bei der bei der Zuweisung eine Kopie erstellt wird, hätte man hingegen überlegen müssen, wie man ein Objekt vom Typ **Zeichnung** in einer Variable vom Typ **Dokument** speichern kann; je nach interner Repräsentation der Objekte (die ja durch den Typ nicht festgelegt ist), ist dafür nämlich gar nicht genug Platz. Gleichzeitig mit der Kopie könnte dann eine Typkonvertierung erfolgen, bei der aus der **Zeichnung** ein Dokument gemacht würde (was auch immer das heißen mag). Dieses Dokument müsste dann, per obiger Typdefinition, auch auf einem **Zeilendrucker** druckbar sein. Es ist allerdings schwer vorstellbar, wie dies umzusetzen ist, wenn das entsprechende Objekt nicht einmal mehr weiß, dass es eine **Zeichnung** ist, geschweige denn, wie seine interne Repräsentation aussieht. In der Praxis der objektorientierten Programmierung ist daher auch nur die Referenzsemantik in Fragen der Zuweisungskompatibilität interessant.

Aliasing und Typeinschränkung

Man beachte übrigens, dass sich bei der Ausgabe aus Methoden (der Rückgabe von Werten) unter Typeinschränkungen kein analoges Problem ergibt: Wenn beispielsweise einem Dokument ein Drucker dauerhaft zugeordnet wird und dieser Drucker mittels einer Methode **drucker** abgefragt werden kann, dann hat die Einschränkung des Rückgabetyps von **drucker** von **Dokument** auf **Plotter** keine negativen Auswirkungen auf die Zuweisungskompatibilität:

kein Problem bei Rückgabe

```
903 | dr <Drucker> |
904 d := z.
905 dr := d drucker
```

ist völlig in Ordnung, solange nur **Plotter** zuweisungskompatibel mit **Drucker** ist. Die unterschiedliche Zulässigkeit von Typeinschränkungen bei Ein- und Ausgabe wird in Abschnitt 26.3 noch genauer beleuchtet.

Was die Freiheit von Typfehlern angeht, kann man das Löschen von Eigenschaften (Methoden) übrigens auch als einen Spezialfall der Typeinschränkung der obigen, zweiten Form auffassen, nämlich einer, in der der Wertebereich auf die leere Menge eingeschränkt wird. So wäre beispielsweise **druckenAuf**: mit einem ParameterTyp ohne Elemente gar nicht mehr aufrufbar (da es kein typkorrektes Parameterobjekt gäbe), was einer Löschung gleichkäme.

Löschen als Spezialfall



26 Subtyping und Inklusionspolymorphie

Die Einführung von Typäquivalenz und Typkonformität bezog sich bislang lediglich auf das Verhältnis der Typdefinitionen, also der Intensionen der Typen. Die Frage des Zusammenhangs der Wertebereiche der Typen, also der Extensionen, ist dabei unberücksichtigt geblieben. Wenn aber die obige Definition von Typkorrektheit weiter Bestand haben soll, dann müssen die Werte zuweisungskompatibler Typen zum Wertebereich des Typen, an den zugewiesen werden sollen, gehören.

Zur Erinnerung: *Typannotationen* stellen *Invarianten* dar, die die möglichen Werte einer Variable beschränken. Diese Invarianten dürfen durch Zuweisungen nicht verletzt werden. Wenn man aber nun Zuweisungen von einem anderen Typen zulässt, dann wird die Typkorrektheit nur dann nicht verletzt, wenn der Wertebereich des anderen Typen (seine Extension) in dem dessen, dem zugewiesen wird, enthalten (inkludiert) ist. Mit anderen Worten: Damit eine Zuweisung `a := b`, bei der sich die Typen von `a` und `b` unterscheiden, zulässig ist, muss die Extension des Typs von `b` eine Teilmenge der Extension des Typs von `a` sein.

Im Fall der Typerweiterung ist dies nicht automatisch der Fall. So handelt es beispielsweise bei der Extension des Typs

mangelnde Teilmengebeziehung bei
Typerweiterung

Typ	DreiDPunkt
erweiterter Typ	ZweiDPunkt
Protokoll	
906	<code>z ^ <Float></code>
907	<code>z: zKoordinate <Float> ^ <Self></code>

als Erweiterung von

Typ	ZweiDPunkt
Protokoll	
908	<code>x ^ <Float></code>
909	<code>x: xKoordinate <Float></code>
910	<code>y ^ <Float></code>
911	<code>y: yKoordinate <Float></code>
912	<code>+ einZweiDPunkt <Self> ^ <Self></code>

nicht unbedingt um eine Teilmenge der Extension von `ZweiDPunkt`, denn es ist z. B. nicht klar, was das Ergebnis der Addition eines dreidimensionalen zu einem zweidimensionalen Punkt sein könnte — geometrisch ist die Addition zweier Punkte unterschiedlicher Dimensionen jedenfalls nicht definiert.



Selbsttestaufgabe 26.1

Versuchen Sie, das Beispiel mit **ZweiDPunkt** und **DreiDPunkt** so zu retten, dass sowohl Typerweiterung als auch Inklusion von Extensionen darin vorkommt. Evtl. finden Sie in Kapitel 9 von Kurseinheit 2 nützliche Hinweise.

Das Phänomen der mangelnden Extensionsinklusion bei Typerweiterung lässt sich darauf zurückführen, dass dem erweiterten Typ (im Beispiel **Dokument**) eigene, d. h. nicht einer seiner Erweiterungen entstammende Werte (Objekte) zugestanden werden. Wäre die Extension eines erweiterten Typs als die Vereinigung der Extensionen seiner Subtypen (hier **Text** und **Zeichnung**) definiert, gäbe es dieses Problem nicht. Dies ist ein sehr guter Grund dafür, dass Supertypen — genau wie Generalisierungen (Kurseinheit 2, Kapitel 9) — keine eigenen Objekte haben sollten (vgl. a. Abschnitt 26.1 und Kurseinheit 7, Kapitel 69).

Herstellung der Teilmengenbeziehung bei Typerweiterung

Auch nicht selbstverständlich ist die Teilmengenbeziehung bei der Typeinschränkung: Durch das Weglassen von Eigenschaften (Methoden) wird die Extension, also die Menge der Werte (Objekte), die darunterfallen, eher größer denn kleiner — je weniger spezifisch die Menge der geforderten Eigenschaften ist, desto mehr Objekte fallen darunter. Die sich daraus ergebende Teilmengenbeziehung wäre also eher die umgekehrte (die Extension des einschränkenden Typen enthält die des eingeschränkten). Etwas anders sieht es aus, wenn durch Typeinschränkung (*Redefinition*) die Ein- oder Rückgabetypen von Methoden beschränkt werden: Die Menge der Zeichnungen ist eine Teilmenge der Menge der Dokumente, auch weil sich Zeichnungen eben nur auf Plottern ausgeben lassen. Die Zuweisungskompatibilität von **Zeichnung** mit **Dokument** wäre also, was die Inklusion der Extensionen angeht, kein Problem.

Teilmengenbeziehung bei Typeinschränkung

Man könnte nun die Typerweiterung unter oben gemachter Einschränkung und die zweite Form der Typeinschränkung als in dieselbe Richtung zielende Maßnahmen ansehen: Beide schränken Extensionen ein. Das lässt sich wie folgt erklären: Wenn man einer Menge von Objekten, die durch eine Anzahl Attribute alle gleichermaßen charakterisiert werden, weitere Attribute beimisst, dann schränkt man diese Menge ein, wenn die hinzugefügten Attribute nicht alle Objekte der Menge charakterisieren. Wenn man beispielsweise wie oben geschehen die Attributmenge des Typs **Dokument** um die Methode **zeilen** ^ <Collection> erweitert, dann fallen die Zeichnungen aus der durch **Dokument** beschriebenen Menge von Objekten heraus, weil sie keine Zeilen haben. Alternativ könnte man auch sagen, dass Dokumente grundsätzlich über Zeilen verfügen können, diese aber bei Zeichnungen immer in der Anzahl 0 vorliegen (also die entsprechende Collection immer leer ist; eine Typeinschränkung!), nur

Erweiterung und Einschränkung als gleichgerichtete Maßnahmen



erscheint das weniger natürlich.⁵¹ Man beachte die Parallelität zum Begriff der *Spezialisierung* (Abschnitt 9.2 in Kurseinheit 2): Der durch Typerweiterung oder -einschränkung aus **Dokument** hervorgegangene Typ **Zeichnung** ist spezieller als seine Vorlage.

Nun ergibt sich aber gemäß obigem Beispiel (Zeilen 900–902) ein Sachverhalt, der trotz aller Harmonie von Typerweiterung und -einschränkung nicht weniger als den Verlust der Zuweisungskompatibilität bedeutet. Dieser resultiert jedoch bei genauerer Betrachtung nicht daraus, dass Zeichnungen keine Dokumente wären, sondern aus der mit der Typkorrektheit verbundenen, impliziten Allquantifiziertheit von Typinvarianten: Eine Methodendeklaration

**Problemquelle
implizite
Allquantifizierung**

913 `druckenAuf: einDrucker <Drucker> ^ <Self>`

im Protokoll eines Typs **Dokument** wird nämlich interpretiert als „**druckenAuf :** ist definiert für alle Empfängerobjekte vom Typ **Dokument** und Parameterobjekte vom Typ **Drucker**“, was aber in dieser Allgemeinheit sachlich falsch ist.

Typsysteme mit Typinvarianten der hier vorgestellten Art sind nicht in der Lage, andere als implizit allquantifizierte Aussagen über Wertebereiche zu treffen. Dies ist gewissermaßen der Preis der Einfachheit. Abhilfe schaffen neuere Typsysteme wie die Idee von den *Dependent types*, wie sie beispielsweise in SCALA zum Einsatz kommen: Hier man sich die Parametertypen von Methoden als Funktionen des Typs, zu dem die Methode gehört, vorstellen. Der Parametertyp von **druckenAuf :** aus obigem Beispiel wäre dann, in Abhängigkeit davon, ob die Methode auf einem Objekt vom Typ **Dokument** oder **Zeichnung** aufgerufen wird, **Drucker** oder **Plotter**. Wie man sich leicht vorstellen kann, ist die statische Prüfung solcher Bedingungen (Invarianten) aber nicht so einfach.

Dependent types

Die Vereinigung von Typerweiterung und Typeinschränkung mit Zuweisungskompatibilität und der daraus folgenden Typkorrektheit bietet der Begriff des Subtyps.

26.1 Der Begriff des Subtyps

Ein **Subtyp** ist als ein Typ definiert, dessen Werte oder Objekte überall da auftauchen dürfen, wo ein Wert des Typs, von dem er ein Subtyp ist, verlangt wird. Subtyp steht dabei nicht für eine besondere Art von Typ, sondern vielmehr für eine Rolle in einer Beziehung zwischen zwei Typen, nämlich der **Subtypenbeziehung**. Die Gegenrolle heißt **Supertyp**.

Man beachte, dass diese Definition von Subtypen Zuweisungskompatibilität impliziert: Wenn die Objekte eines Subtypen überall da auftauchen dürfen, wo Objekte seines Supertypen erwartet werden, dann dürfen sie auch Werte von

Zuweisungskompatibilität von Subtypen

⁵¹ Stattdessen würde man eher vermuten, dass es sich um einen Programmierfehler handelt, wenn jemand bei einer Zeichnung auf ihre Zeilen zugreifen will. Außerdem müsste bei erster Annahme der allgemeinste Typ, von dem alle anderen abgeleitet sind (**Object** in STRONGTALK), immer alle Attribute deklarieren, die einem jemals in den Sinn kämen, und das wäre nun wirklich unpraktisch.



Variablen sein, die mit dem Supertypen annotiert (auf Werte des Supertypen beschränkt) sind. Ein Subtyp ist also mit seinem Supertyp per Definition zuweisungskompatibel. Es steckt in dieser Definition aber eine gewisse Zirkularität (Subtyp als Voraussetzung und Ergebnis der Zuweisungskompatibilität), die eine Einfachheit der Zusammenhänge vortäuscht, die es in Wirklichkeit nicht gibt; die eigentliche Frage, was nämlich erfüllt sein muss, damit ein Objekt eines Typen tatsächlich da erscheinen darf, wo ein Objekt eines anderen Typen erwartet wird, bleibt unberücksichtigt. Eine Befassung mit dieser Frage erfolgt hier aber nur insoweit, wie dies heutige Typsysteme auch tatsächlich tun; eine genauere Betrachtung erfolgt dann erst in Kurseinheit 6, Kapitel 54.

Ein Subtyp kann selbst wieder Subtypen haben usw.; man spricht dann

Subtyp hierarchie

auch von einer **Subtypen-** oder einfach nur von einer **Typhierarchie**. In einer solchen Hierarchie kann man **direkte** von **indirekten Subtypen** unterscheiden: Zwischen einem Typ und seinem direkten Subtyp liegt kein weiterer Typ in der Typhierarchie, bei einem indirekten Subtyp hingegen schon. Die Subtypenbeziehung ist transitiv und reflexiv; insbesondere ist also jeder Typ ein Subtyp von sich selbst (das folgt schon aus obiger Definition des Begriffs Subtyp). Die Frage der Symmetrieeigenschaft muss noch bis zum nächsten Abschnitt zurückgestellt werden.

Je nach verwendetem Typsystem kann ein Typ auch mehrere direkte Supertypen haben. Die sich daraus ergebende Struktur ist dann aber keine

mehrere direkte Supertypen

Hierarchie mehr (im strengen Sinne; man spricht aber dennoch häufig von einer solchen, manchmal auch von einer **Mehrfachhierarchie**), sondern nur noch ein gerichteter azyklischer Graph (engl. directed acyclic graph, kurz DAG). Alle obengenannten Eigenschaften der Subtypenbeziehung bestehen jedoch weiter fort.

Wenn Subtypen, ähnlich wie bei der Typerweiterung oder -einschränkung, auf Basis von bereits bestehenden definiert werden, spricht man

Begriff des Subtyping

auch vom (nominalen) **Subtyping** (s. u.). Eine solche Subtypendefinition erfolgt dann immer unter Angabe des oder der direkten Supertypen, und relativ dazu. Dabei verlangt die obige Definition von einem Subtypen einen bestimmten Zusammenhang zwischen den Definitionen (Intensionen) von Sub- und Supertyp: Die Ergänzungen oder Änderungen, die eine Subtypendefinition relativ zu der ihres oder ihrer Supertypen vornimmt, müssen gewährleisten, dass die Werte (Objekte) des Subtyps überall da auftauchen dürfen, wo ein Wert des Supertyps verlangt wird. Dies lässt sich durch folgende einfache Regel ausdrücken:

Wenn ein Typ Y ein Subtyp eines Typs X ist, dann müssen alle Bedingungen, die für Objekte des Typs X erfüllt sind, auch für Objekte des Typs Y erfüllt sein.

Es darf also insbesondere keine Bedingung, die ein Supertyp an seine Objekte stellt, durch einen Subtyp aufgehoben oder relativiert werden. Logisch gesprochen heißt das, dass die Bedingungen (die Intension) des Subtyps die des Supertyps impliziert. Daraus folgt, dass die



Typweiterung als Basis einer Subtypendefinition infrage kommt (da die Intension des Supertypen unverändert übernommen und lediglich ergänzt wird), die Typeinschränkung hingegen zunächst einmal nicht. Dennoch wäre die Typeinschränkung vom Subtyping auszuschließen eine unnötige Einschränkung, wie Sie gleich noch sehen werden.

Wenn man die eingangs dieses Kapitels gemachten Bemerkungen zur Typkorrektheit auf das Subtyping und die damit implizierte Zuweisungskompatibilität überträgt, dann ergibt sich für die Extensionen von Supertypen und Subtypen, dass die Subtypenbeziehung als eine Teilmengenbeziehung gedeutet werden muss: Die Extension eines Subtyps ist in den Extensionen all seiner (direkten und indirekten) Supertypen enthalten. Umgekehrt umfasst die Extension eines Supertyps die Extensionen all seiner Subtypen. Es ergibt sich, dass nur wenn die Extensionen aller direkten Subtypen eines Typs paarweise disjunkt sind, man es mit einer echten Typhierarchie zu tun hat, in der jeder Typ nur genau einen direkten Supertyp hat. Ist die Extension eines Supertyps genau gleich der Vereinigung der Extensionen seiner Subtypen, hat der Supertyp keine eigenen Werte, also keine Werte, die nicht zugleich Wert eines seiner Subtypen sind. Diese Bedingung entspricht der Idee von der *Generalisierung* aus Abschnitt 9.1 und im übrigen gute objektorientierter Praxis (s. Kapitel 69).

Subtypen- als Teilmengenbeziehung

26.2 Strukturelles und nominales Subtyping

Beim Subtyping unterscheidet man wie bei der Typäquivalenz und -konformität zwischen nominalem und strukturellem Subtyping. Nominales Subtyping liegt vor, wenn ein Subtyp aus einem namentlich erwähnten Supertyp abgeleitet sein muss, um als sein Subtyp zu gelten. Strukturelles Subtyping liegt vor, wenn ein Typ lediglich die obige Definition von Subtyp erfüllen muss, um als solcher zu gelten. Nominales Subtyping impliziert strukturelles; analog zur Typkonformität macht das nominale Subtyping die Subtypenbeziehung antisymmetrisch, das strukturelle hingegen nicht.

26.3 Kovarianz und Kontravarianz bei Methodenaufrufen

Dass Typweiterung als Basis des Subtyping keine technischen Probleme bereitet, sollte hinreichend klargeworden sein: Typfehler sind damit ausgeschlossen und es bleibt lediglich das semantische Problem, dass Werte eines Subtyps inhaltlich keine Werte des Supertyps sind (wie im Beispiel von zwei- und dreidimensionalen Punkten). Es bleibt noch die Frage, ob und falls ja in welchem Umfang Typeinschränkung im Rahmen des Subtyping erlaubt ist. Diese Frage soll an einem Beispiel beantwortet werden.

Angenommen, es ist ein Typ A wie folgt definiert:

Typ	A
Protokoll	



Zuweisungen der Art

915 x := a m: z

wobei **a** eine Variable vom Typ A sei (für ein Objekt vom Typ A steht), sind nach den Regeln des Subtyping dann zulässig, wenn die Variable **x** mit einem Supertyp und **z** mit einem Subtyp von Y (jeweils einschließlich des Typs Y selbst) deklariert ist.

Nun sei weiterhin der Typ B wie folgt als Subtyp von A abgeleitet:

Typ	B
Supertyp	A
Protokoll	

916 m: y <Z> ^ <X>

Die Methode **m:** wird also in **B** *redefiniert*. Die Frage ist nun, in welchem Verhältnis die dabei verwendeten Typen X und Z zu Y stehen müssen, damit die Zuweisung aus Zeile 915 weiterhin zulässig ist, selbst wenn die Variable **a** auf ein Objekt vom Typ B verweist. Mit anderen Worten: Welche Bedingungen sind an die Parametertypen bei der Redefinition zu stellen, damit eine Zuweisung eines Objekts vom Typ B an eine Variable vom Typ A in der Folge zu keiner Verletzung einer (anderen) Typinvariante führt? Solche Folgefehler waren ja bereits in Kapitel 25 thematisiert worden.

Die Antwort lässt sich systematisch herleiten, indem man sich die zu Zeile 915 gehörenden impliziten Zuweisungen genau anschaut. Zuerst wird ja **z** dem formalen Parameter von **m:**, **y**, zugewiesen. Wenn **y** nun in **B** einen anderen Typ als **Y** bekommen soll, dann darf es sich dabei nur um einen handeln, der mehr Werte zulässt — würde er weniger Werte zulassen, könnte es sein, dass er die Zuweisung von **z** ausschließt, wodurch Zeile 915 zu einem Typfehler (einem typinkorrekten Programm) führen würde. Der Typ des formalen Parameters **y** von **m:** in **B**, **Z**, muss also ein Supertyp dessen in **A**, **Y**, sein. Zuletzt, d. h., nach erfolgter Auswertung des Methodenaufrufs, wird dann das Ergebnis **x** zugewiesen. Wenn nun der Rückgabewert von **m:** in **B**, **X**, einen anderen Typ als in **A**, **Y**, bekommen soll, dann kann dies aufgrund der geforderten Zuweisungskompatibilität nur um einen Typen handeln, der weniger Werte zulässt, da ja sonst die Zuweisung an **x** zu einem Typfehler führen könnte. Es kommt also als Rückgabetyp für **m:** in **B** nur ein Subtyp von dem in **A**, **Y**, infrage. Es ergibt sich also, dass sich bei einer Redefinition einer Methode die Eingabeparametertypen einer Funktion nur „nach oben“ (also zu einem Supertypen hin), die Ausgabeparameter hingegen nur „nach unten“ (hin zu einem Subtyp) verändern dürfen, wenn die Typkorrektheit eines Programms nicht verletzt werden soll.

**analytische
Betrachtung**



Nun ändern sich aber bei der Redefinition nicht nur die Parametertypen (Ein- und Aus- bzw. Rückgabe), sondern auch der Typ des Empfängers. Dieser ändert sich bei der Redefinition aber immer nach unten (da der redefinierende Typ ja als Subtyp vom redefinierten abgeleitet wird). Es folgt also, dass die Eingabeparameter zum Empfängertyp gegenläufig variieren müssen, der Ausgabeparameter hingegen gleichgerichtet. Man spricht im ersten Fall daher von einer **Kontravarianz**, im zweiten von einer **Kovarianz**, und sagt:

Wenn die Eingabeparameter einer redefinierten Methode kontravariant und die Ausgabeparameter kovariant redefiniert werden, dann bleibt Zuweisungskompatibilität des redefinierenden Typen mit dem redefinierten erhalten.

Man spricht im Kontext von Subtyping auch von *Typkonformität des Subtypen mit dem Supertypen*.

Nun enthält die Idee von der Gegenläufigkeit der Veränderung von Parameter- und Ergebnistypen beim Redefinieren einen kleinen Schönheitsfehler: Wenn es sich nämlich bei der Eingabe in eine Funktion und bei ihrer Ausgabe um dasselbe Objekt handelt, kann diesem nicht einmal (bei der Eingabe) ein Supertyp und einmal (bei der Ausgabe) ein Subtyp zugeschrieben werden, denn der Subtyp verlangt ja mehr Eigenschaften, als der Supertyp garantiert. Wenn also z. B. ein Typ A wie

**Problem bei Identität
von Ein- und
Ausgabeparametern**

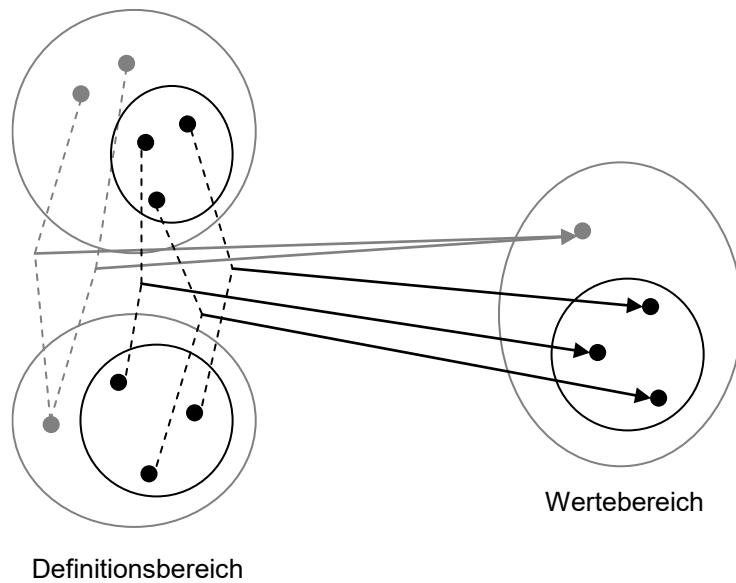
Typ | A
Protokoll |
917 x ^ <X>
918 x: <X> ^ <Selbst>

definiert ist und x das Objekt zurückgeben soll, das mit x: geliefert wurde, dann kann ein Subtyp B die beiden Methoden schlecht so redefinieren, dass x: Objekte eines Supertypen von X entgegennimmt, während gleichzeitig x Objekte eines Subtyps zurückliefert. Da das Umgekehrte freilich auch nicht geht, bleibt nichts anderes, als die Unveränderlichkeit der Parametertypen, auch als **Invarianz** oder besser (da das dazu passende Adjektiv „invariant“ in seiner Bedeutung schon belegt ist) als **Novarianz** bezeichnet, zu verlangen.

Obige analytischen Überlegungen führen also, mit der eben gemachten Einschränkung, zur Regel von den kontravarianten Parameter- und den kovarianten Rückgabetypredefinitionen. Es gibt aber noch einen zweiten Ansatz zur Klärung der Frage nach der richtigen Varianz der Parametertypen redefinierter Funktionen, die diese Betrachtung deutlich in Frage stellen. Dazu soll (wie schon in Kapitel 25) die Interpretation von Methoden als Relationen bzw., da hier der Rückgabetyp mit be-

**Problem des
mangelnden
Realismus**

rücksichtigt wird, als Funktionen oder Abbildungen herhalten. Der hier zweistellige Definitionsbereich (Empfängertyp plus Parametertyp) der Funktion steht dabei stellvertretend für beliebige Stellenzahl, also für Methoden mit beliebig vielen Parametern. Der Wertebereich ist hingegen immer einstellig, da eine Methode stets nur einen Wert zurückgibt.



Wenn man nun die Anzahl der Empfängerobjekte einschränkt (was ja beim Übergang zu einem Subtypen geschieht), dann schrumpft damit nicht nur der Wertebereich der Funktion (wie in Kapitel 25 schon illustriert), sondern auch die Menge der möglichen Eingabewerte (der zweite und alle weiteren Definitionsbereiche), die mit der bereits eingeschränkten Menge der Empfänger gemeinsam auftreten können. Es verhalten sich also nicht nur die Ergebnis-, sondern auch die Parametertypen kovariant.

Dieses Ergebnis ist gewissermaßen frustrierend, da es die soeben hergeleitete Kontravarianzregel für Parametertypen infrage stellt: Was programmiertechnisch möglich und sinnvoll erscheint, hat in der Realität (der Interpretation oder Semantik) keine Bedeutung. Auf der anderen Seite erklärt es aber, warum kontravariante Parameterredefinitionen in der Programmierpraxis nicht benötigt werden.⁵² Kovarianz für Parametertypen zuzulassen, so sinnvoll es auch zu sein scheint, erlaubt jedoch typinkorrekte Programme; Sie werden im Kontext der Programmiersprache EIFFEL (Kurseinheit 5, Abschnitt 52.5) noch ausführlicher auf das Problem und eine mögliche Lösung hingewiesen.

26.4 Inklusionspolymorphie

Ein von Christopher Strachey, einem der Urväter der Programmierung als wissenschaftliche Disziplin, eingeführter Begriff ist der der **Polymorphie**. Polymorphie bedeutet allgemein

⁵² Wer ein Beispiel weiß oder hat, möge es mir bitte schicken!



Vielgestaltigkeit und wird vor allem in der Biologie verwendet. In der Programmierung steht er für verschiedene Dinge, die jedoch alle mit Typen zu tun haben.

Unter **Inklusionspolymorphie**, auch **Subtyppolymorphie** genannt, versteht man im wesentlichen dasselbe wie unter Subtyping: Wo Objekte eines Typs erwartet werden, können Objekte anderer Typen erscheinen, weil der erste Typ die anderen subsumiert (inkludiert). Der Begriff ist vor allem in Abgrenzung zum *parametrischen Polymorphismus* (engl. parametric polymorphism, s. Kapitel 29) gebräuchlich; sonst redet man eher von Subtyping.

Das Interessante an der Inklusionspolymorphie ist, dass sich der Wertebereich von Typen dadurch auf unvorhergesehene Umfänge aufweiten lässt. Dies ist insbesondere für die Weiterentwicklung und Wiederverwendung von Programmen interessant, bei der einfach neue Typen hinzugefügt, die anstelle bereits existierender eingesetzt werden können, ohne dass dazu am Programm sonst etwas geändert werden müsste. Die Regeln einer strengen Typprüfung werden durch Inklusionspolymorphie aufgelockert, ohne an Typsicherheit zu verlieren.

Attraktivität der Inklusionspolymorphie

Insgesamt krankt die Definition des Subtyping und der Inklusionspolymorphie in der objektorientierten Programmierung jedoch daran, dass nicht klar definiert ist, was alles zu verlangen ist, damit ein Objekt eines Typs tatsächlich auch da auftauchen kann, wo ein Objekt eines anderen Typen erwartet wird. Zwar gibt die Regel von Ko- und Kontravarianz eine klare Bedingung vor, aber wie Sie schon gesehen haben, ist diese Bedingung aus praktischen Gründen nicht unumstritten. Dazu kommt, dass die Regel einerseits gar nicht ausreicht, um Ersetzbarkeit zu garantieren, und andererseits zu streng ist (s. Kurseinheit 6, Kapitel 54). Da Ersetzbarkeit aber der Definition des Subtypenbegriffs zugrunde liegt, bleibt das ganze schwammig. In dieser Kurseinheit habe ich mich, den meisten gängigen objektorientierten Programmiersprachen folgend, darauf zurückgezogen, zu garantieren, dass keine Typfehler, also Fehler der Art, dass eine bestimmte, geforderte Eigenschaft (Methode) bei einem Objekt nicht vorhanden ist, auftreten können; alles weitere wird dann in Kapitel 54 behandelt.

Unzulänglichkeit der einfachen Definition

27 Typumwandlungen

Zuweisungskompatibilität unter Subtyping erlaubt also die Zuweisung von Objekten eines Subtyps an Variablen eines Supertyps. Für die statische Typprüfung ergibt sich daraus kein Problem, weil sichergestellt ist, dass die Subtypen alle Eigenschaften ihrer Supertypen erhalten, so dass keine Typfehler auftreten können. Für die Programmiererin ergibt sich aber manchmal das Problem, dass sie ein Objekt, auf das eine Variable eines Supertyps verweist, wie ein Objekt seines tatsächlichen Typs verwenden möchte, in der Regel, weil sie eine Methode darauf aufrufen möchte, die der Supertyp nicht hat. Genau diesen Methodenaufruf würde die Typprüfung aber zurückweisen.



Für diesen Zweck gibt es die Möglichkeit der **Typumwandlung** (engl. type cast). Eine Typumwandlung ist ein Verfahren, bei dem der vorgefundene Typ eines Ausdrucks (einer Variable oder eines Methodenaufrufs) in einen vorgegebenen konvertiert wird. Mit dem Objekt, für das der Ausdruck steht, passiert dabei gar nichts — es wird lediglich der Compiler (bzw. der Type checker) davon überzeugt, dass der Ausdruck den bei der Umwandlung angegebenen Typ hat. Sollte sich zur Laufzeit herausstellen, dass das nicht der Fall ist, kann ein Laufzeittypsystem — soweit vorhanden — dies bei seiner *dynamischen Typprüfung* bemerken und ggf. einen entsprechenden Fehler melden (vgl. die Anmerkungen dazu in Kapitel 18).

Typumwandlungen können grundsätzlich in verschiedene Richtungen erfolgen: zu Supertypen, zu Subtypen oder zu solchen, die weder Super- noch Subtyp des Ausgangstyps sind. Man spricht entsprechend von **Up cast**, **Down cast** oder **Cross cast**. Up casts sind immer typsicher, Down casts und Cross casts nicht. Down casts sind relativ häufig; sie kommen vor allem dort vor, wo kein parametrischer Polymorphismus (Kapitel 29) zur Verfügung steht oder wo ein Objekt seinem tatsächlichen Typ entsprechend behandelt werden soll. Cross casts sind eher selten; in der *interfacebasierten Programmierung* (s. Abschnitt 28.2) stehen sie für einen *Rollenwechsel eines Objekts*.

Arten von

Typumwandlungen

Nun sind Typumwandlungen entweder überflüssig oder unsicher. Man sollte daher versuchen, auf sie zu verzichten. Wo unverzichtbar, sollten

Verwendungs- empfehlungen

Typumwandlungen mit einem Typtest abgesichert werden. Dabei wird zur Laufzeit geprüft, ob das Objekt, für das der typgewandelte Ausdruck steht, auch den gewünschten Typ hat. Ist das nicht der Fall, sollten die Teile des Programms, die den bei der Typumwandlung genannten Typ voraussetzen, nicht ausgeführt werden. Sie werden in späteren Kapiteln zu den einzelnen Programmiersprachen noch Beispiele für diese Praxis zu sehen bekommen.

28 Der Zusammenhang von Typen und Klassen

Wenn in dieser Kurseinheit bislang ausschließlich von Typen die Rede war und Klassen dabei ignoriert wurden, so hat das gute Gründe: Während eine Klasse die Implementierung ihrer Objekte festlegt, ist eine Typdefinition vollkommen frei von Implementierungsaspekten. Zwar können auch *abstrakte Klassen* (Kurseinheit 2, Abschnitt 10.3) ausschließlich aus Methodendeklarationen bestehen, also ohne jeden Implementierungsanteil daherkommen, aber auch ihr Zweck ist in der Regel, zumindest eine partielle Implementierung vorzugeben, die anderen Klassen, ihren Subklassen, gemeinsam ist, so dass sie diese erben können: Schließlich drückt die Klassenhierarchie ja eine „genetische“ Verwandtschaft aus (s. Abschnitt 10.1 und Kapitel 11). Eine Typprüfung soll aber ohne Ansehen der Implementierung stattfinden; sie baut daher auf abstrakte Spezifikationen, eben auf Typen.

Es sind also Typen abstrakte Spezifikationen, die zum einen den Wertebereich von Variablen einschränken und zum anderen das *Protokoll* (den Funktionsumfang) von Objekten ange-



ben. Im Gegensatz dazu sind Klassen Konstrukte, die Objekte als Instanzen zu bilden erlauben und mit Implementierung versehen. Da Objekte aber auch den Wertebereich von Typen ausmachen, stellt sich natürlich die Frage, welcher Art der Zusammenhang zwischen Typen und Klassen ist.

Diese Frage soll anhand der schematischen Klassendefinitionen aus Kurseinheit 2, Abschnitt 7.2 beantwortet werden. In SMALLTALK ist diese ja stets von der Form⁵³

```
Klasse          <Klasse 1>
Superklasse     <Klasse 2>
benannte Instanzvariablen <Instanzvariable 1>, ...
Instanzmethoden |  
919  <Instanzmethode 1>: <formaler Parameter 1> ...
920  ...
```

Es fällt zunächst auf, dass bestimmte Elemente einer Klassendefinition auch in einer Typdefinition auftauchen. Im einzelnen sind dies

Vergleich von Klassen- und Typdefinition

- ein (eindeutiger) Name,
- ein zweiter Name, von dessen dazugehöriger Definition abgeleitet wird sowie
- eine Menge von Methodennamen, jeweils mit einer Anzahl formaler Parameter.

Nun werden in SMALLTALKs Klassendefinitionen anders als bei den Typdefinitionen STRONGTALKs keine Typen verwendet — wie auch, denn in SMALLTALK gibt es ja schließlich keine Typen. Stattdessen findet man aber in SMALLTALK-Programmen manchmal Namen wie „aString“, „anInteger“ etc. für formale Parameter, die nahelegen, dass der Wert einer Variable Instanz einer bestimmten Klasse sein soll. Überprüft wird das jedoch nicht. In STRONGTALK hingegen ist die Ähnlichkeit von Klassendefinitionen mit Typdefinitionen noch größer: Hier sind auch die formalen Parameter der Methoden in den Klassendefinitionen typisiert (s. Kapitel 20). Man beachte, dass in STRONGTALK, anders als z. B. in JAVA oder C++, Instanzvariablen kein Bestandteil einer Typdefinition sein können (vgl. dazu Fußnote 50).

Nun dient ja ein Typsystem in der objektorientierten Programmierung vor allem der Sicherstellung des Umstands, dass alle von einem Objekt aufgrund des deklarierten Typs der Variable, die es benennt, erwarteten Eigenschaften (Methoden) bei diesem Objekt auch vorhanden sind. Dies ist aber immer dann der Fall, wenn sich die Elemente der Typdefinition in der Klassendefinition des Objekts wiederfinden, die Klassendefinition also mit der Typdefinition gewissermaßen strukturell konform ist, so dass die Zuweisung einer Instanz der Klasse an eine Variable des Typs die Anforderungen der Zuweisungskompatibilität erfüllt. Um einen

⁵³ KlassenvARIABLEN und -METHODEN können hier unter den Tisch fallen, da diese ja nicht Objekte, sondern Klassen (als Instanzen ihrer Metaklassen) charakterisieren.



Compiler diese Zuweisungskompatibilität auf einfachere Weise als die Prüfung der Strukturkonformität, die ja eine *rekursive Expansion* der Typdefinitionen erfordert, feststellen zu lassen, gibt es zwei Möglichkeiten (bei beiden handelt es sich gewissermaßen um Varianten einer Namenskonformität):

1. jede Klasse sagt explizit, mit welchen Typen sie konform ist, oder
2. jede Klasse spezifiziert implizit selbst einen Typ.

Im ersten Fall müsste der Compiler noch prüfen, ob eine Klasse tatsächlich auch über alle Eigenschaften der von ihr genannten Typen verfügt; im zweiten Fall ist das automatisch der Fall, da der Typ ja gewissermaßen aus der Klasse erzeugt wird. Diese zweite Art wird von den allermeisten typisierten, objektorientierten Programmiersprachen bevorzugt, doch auch die erste kommt in populären Sprachen vor: So kann beispielsweise in JAVA und C# jede Klasse angeben, mit Variablen welcher Interface-Typen ihre Instanzen zuweisungskompatibel sein sollen (s. Kurseinheit 4, Kapitel 40 und Kurseinheit 5, Abschnitt 50.4.2). Auch STRONGTALK stellt beide Möglichkeiten zur Verfügung.

28.1 Subklassen und Subtypen

Man könnte nun versucht sein, den Zusammenhang von Klassen und Typen auch unter Vererbung bzw. Subtyping beizubehalten und damit zu erwarten, dass eine Instanz einer Subklasse einer Klasse dem Wertebereich des zur Superklasse gehörenden Typs angehört. Das ist jedoch dann nicht der Fall, wenn in der Subklasse Änderungen vorgenommen werden, die eine Typkonformität vom zur Subklasse gehörendem zum zur Superklasse gehörenden Typ aufheben, also z. B. Methoden gelöscht oder inkompatibel redefiniert werden. Die meisten objektorientierten Programmiersprachen verbieten das jedoch, so dass sich die Subklassenbeziehung tatsächlich auf eine parallele Subtypenbeziehung übertragen lässt.

28.2 Typen als Schnittstellenspezifikationen von Klassen

Eine Klasse liefert eine Implementierung. Nach gängigen Prinzipien nicht nur der objektorientierten Programmierung sind Implementierungen aber hinter *Schnittstellen* (oder *Interfaces*) zu verbergen: Nur die Elemente einer Klassendefinition, die für Benutzerinnen einer Klasse zu Verwendung gedacht sind, sollen durch die Schnittstelle nach außen getragen werden — der Rest soll verborgen bleiben (das sog. *Geheimnisprinzip*).

In Programmiersprachen wie JAVA, C++ etc. gibt es spezielle Schlüsselwörter, die einem Element einer Klassendefinition (beispielsweise einer Methode) vorangestellt seine Zugreifbarkeit festlegen. Diese sog. *Zugriffsmodifikatoren* (engl. access modifier) legen gemeinsam mit der Klassendefinition, die ihre vollständige Implementierung beinhaltet, auch die Schnittstelle der Klasse fest. Je nach Sprache ist diese Schnittstelle für alle Benutzerinnen der Klasse gleich oder unterscheidet sich nach Lokalität oder anderen Eigenschaften von benutzender und benutzter Klasse. Im

Zugriffsmodifikatoren; absolute und relative Schnittstelle



ersten Fall könnte man von einer absoluten Schnittstelle sprechen; um sie zu spezifizieren, reicht es, zwischen sichtbar und unsichtbar zu unterscheiden. Im zweiten Fall ist die Schnittstelle relativ.

Eine absolut spezifizierte Schnittstelle einer Klasse kommt, wenn sie wirklich keinerlei Implementierungsgeheimnisse verrät, einem Typ gleich. Sie besteht nämlich nur aus Deklarationen von Methoden. Die gemachte Einschränkung ist notwendig, weil manche Sprachen, so z. B. JAVA und C++, die Instanzvariablen ihrer Objekte in die Schnittstelle der Klassen aufzunehmen erlauben. Mit den Instanzvariablen wird aber die Repräsentation der Objekte nach außen sichtbar, was dem Gedanken des Geheimnisprinzips widerspricht.

absolute
Schnittstellen als
Typen

Wenn man nun eine Variable mit einem solchen die Schnittstelle repräsentierenden Typ deklariert und eine Typprüfung erfolgreich durchgeführt hat, dann ist sichergestellt, dass über diese Variable nur auf die Elemente einer Klasse zugegriffen wird, die auch Bestandteil des Interfaces der Klasse sind. Wenn jede Instanz dieser Klasse ausschließlich über typisierte Variablen ansprechbar ist, ist damit die Wahrung des Geheimnisprinzips garantiert. Typen dienen damit einem weiteren Zweck, den man zunächst einmal nicht mit ihnen assoziieren würde, nämlich der Wahrung des Implementationsgeheimnisses/Einhaltung der Schnittstellen durch den Compiler.

Wahrung des
Implementationsge-
heimnisses

Dieser überaus nützliche Zusammenhang zwischen Klassen, ihren Schnittstellen und Typen wurde erst relativ spät, nämlich mit der Programmiersprache JAVA und ihrem *Interface-als-Typ-Konzept*, so weiterentwickelt, dass eine Klasse verschiedene Schnittstellen anbieten kann, die alle zugleich Typen der Klasse (genauer: Supertypen des der Klasse entsprechenden Typs) sind. Die damit ermöglichte *interfacebasierte Programmierung*, die in Kurs 01853 ausführlich behandelt wird, betrachte ich persönlich als den wichtigsten Beitrag JAVAs zur Disziplin der objektorientierten Programmierung (s. a. Kurseinheit 4, Kapitel 45).



28.3 Gründe für die Trennung von Typen und Klassen

Nun mögen Sie sich vielleicht fragen, warum Typen und Klassen über so viele Seiten als getrennte Begriffe dargestellt wurden, nur um am Ende zum Schluss zu kommen, dass eine Klassendefinition in der Regel auch als Typdefinition herhält. Nun, erstens ist das nicht in allen Sprachen der Fall und zweitens ist es selbst in den Sprachen, in denen es der Fall zu sein scheint, nicht immer so (s. Fußnote 54). So handelt es sich eher um die Symbiose zweier verschiedener Konzepte, die unterschiedlichen Zwecken dienen, deren strukturelle Ähnlichkeit sich aber durch eine syntaktische Zusammenlegung ausnutzen lässt:

1. Klassen dienen der Angabe von Implementierungen und damit als Container von ausführbarem Code;
2. Typen dienen der Formulierung von Invarianten, die für Variablenbelegungen gelten müssen und deren Verletzung auf einen (logischen oder semantischen) Programmierfehler hinweist.



Da beide im wesentlichen über die gleichen Elemente verfügen, lässt sich die Definition beider in einem Sprachkonstrukt zusammenfassen.

Der Unterschied der beiden Konzepte Klasse und Typ manifestiert sich auch darin, welche Rolle sie zur Laufzeit eines Programms spielen: Typinformation beeinflusst die Ausführung eines laufenden Programms insofern, als sie ein Programm bei Verletzung einer Invariante abbrechen lässt (durch einen dynamischen Typtest) und damit einem anderen, schwieriger zuordenbaren Fehler zuvorkommt. Klasseninformation beeinflusst die Ausführung des laufenden Programms insofern, als sie Grundlage des dynamischen Bindens ist und in einem Programm als Eigenschaft von Objekten abgefragt werden kann. In Sprachen, in denen jede Klasse einen Typ definiert, ist diese Unterscheidung jedoch nicht immer klar getroffen und wird deswegen von Programmiererinnen auch nicht unbedingt wahrgenommen.

Unterschiede zur Laufzeit

29 Generische Typen oder parametrischer Polymorphismus

Typen beschränken die Wertebereiche von Variablen und Methoden. Inklusionspolymorphie lockert diese Beschränkung insofern, als dadurch Wertebereiche von Typen um die von Subtypen erweitert werden können, selbst wenn diese Subtypen zum Zeitpunkt der Typdefinition noch gar nicht bekannt waren (Abschnitt 26.4). Nun ist Inklusionspolymorphie nicht die einzige Möglichkeit, den Wertebereich eines Typs variabel zu halten, ohne die statische Typprüfung aufzugeben zu müssen. Eine andere ist, einen Typ mit einem oder mehreren anderen zu parametrisieren.

Eine **parametrische Typdefinition** unterscheidet sich von einer normalen dadurch, dass in der Typdefinition verwendete, andere Typen nicht genannt (referenziert) werden müssen, sondern durch Platzhalter, die Typparameter, vertreten werden können. Diese Platzhalter sind Variablen, deren Wert implizit (also ohne entsprechende Deklaration) auf Typen beschränkt ist; man nennt sie auch **Typvariablen**. Diese Typvariablen werden erst bei der Verwendung eines parametrisierten Typs in der Deklaration eines anderen Programmelements mit einem Wert, also einem Typ, belegt. Man spricht bei dieser Wertzuweisung an eine Typvariable von einer **Instanziierung des parametrischen Typs**; erst bei ihr entsteht ein konkreter Wertebereich, der dann dem deklarierten Programmelement zugeordnet wird. Insbesondere hat ein parametrischer Typ, bei dem Typvariablen nicht belegt sind, keinen konkreten Wertebereich. Dieser Umstand ist bei der Betrachtung von *Zuweisungskompatibilität unter parametrischem Polymorphismus* wichtig.

parametrische Typdefinition; Instanziierung

Die Idee des **parametrischen Polymorphismus** ist, aus einer Typdefinition durch Parametrisierung viele zu machen. Eine parametrische Typdefinition steht also nicht für einen Typ, sondern für (theoretisch) beliebig viele — sie erlaubt es gewissermaßen,

generische Typen



Typen nach Bedarf zu generieren.⁵⁴ Wohl deswegen bezeichnet man parametrische Typen (Typdefinitionen) auch als **generische Typen** (Typdefinitionen) oder kurz als **Generics**. Wie eben schon erwähnt, wird der Wertebereich bei einer solchen Typgeneration jeweils mitgeneriert.

Es erfolgt also die Zuweisung eines Typs zu einer Typvariable bei der Verwendung eines parametrisch definierten Typs in einer Deklaration, beispielsweise der Deklaration einer Variable oder des Rückgabewerts einer

**formale und
tatsächliche
Typparameter**

Methode. Oberflächlich betrachtet entspricht diese Verwendung in etwa dem Aufruf einer (ja auch an einer anderen Stelle definierten) Methode oder besser (und schon aufgrund der Verwendung des Begriffs Instanziierung) eines Konstruktors; deswegen nennt man die Typvariablen, die in parametrischen Typdefinitionen vorkommen, auch **formale Typparameter** und die konkreten Typen, die bei der Verwendung des Typen in Deklarationen in die formalen Parameter eingesetzt werden, auch **tatsächliche Typparameter**. Trotz dieser Analogie zu Methoden- bzw. Konstruktoraufrufen muss man sich immer vor Augen halten, dass die Verwendung eines parametrisch definierten Typs bereits zur Übersetzungszeit zu einer Zuweisung an die Typvariablen führt, man es also keineswegs mit etwas Dynamischem zu tun hat. Insbesondere müssen Typen keine Objekte sein, um Typvariablen zugewiesen werden zu können.

29.1 Einfacher parametrischer Polymorphismus

Ein einfaches Beispiel für eine generische Typdefinition in STRONGTALK ist das folgende:

Typ]	A
Typvariablen]	T
Protokoll]	
921	x ^ <T>
922	x: einT <T> ^ <Self>

T ist dabei eine Typvariable. Beim Vorkommen von T im Abschnitt „Typvariablen“ handelt es sich um ihre Deklaration (Vereinbarung); beim Vorkommen im Abschnitt „Protokoll“ um ihre Verwendung.

Das für den Tatbestand der Parametrisierung wichtige an dieser Typdefinition ist, dass x: anstelle des Parameter- und x anstelle des Rückgabetyps T nennt, wobei T eben kein Typ, sondern eine Typvariable ist. Für Typvariablen verwendet man traditionell einzelne Großbuchstaben; dies hat den nützlichen Nebeneffekt, dass man durch eine Typvariable keinen tatsächlichen Typen verdeckt, wie es sonst versehentlich passieren könnte: Man könnte die Typvariable nämlich auch beispielsweise „Integer“ nennen, aber sie wäre deswegen immer

⁵⁴ Entsprechend gehören, einen Zusammenhang von Klassen und Typen wie in Abschnitt 28 beschrieben vorausgesetzt, zu einer parametrischen Klassendefinition beliebig viele Typen.



noch eine Variable und der Typ **Integer** wäre innerhalb der Typdefinition nicht mehr sichtbar.

Wenn man nun den Typ A verwenden, also z. B. eine temporäre Variable vom Typ A deklarieren möchte, muss man sich festlegen, welchen Wert die Typvariable T in der Typdefinition und damit welchen Typ die Rückgabe von x und die Eingabe von x: haben sollen. Soll T beispielsweise den Wert **Integer** bekommen, dann schreibt man

beispielhafte Instanziierung

```
923 | a <A[Integer]> |
```

und *instanziert* dabei den parametrischen Typen. **Integer** ist dabei der tatsächliche Typparameter (eine Typkonstante, wenn man so will), der in STRONGTALK in eckige Klammern gesetzt wird. Er wird für diese Verwendung des parametrischen Typs (und nur für diese) in den formalen Typparameter (die Typvariable) eingesetzt. Der Typ von a, A[**Integer**], wird damit zu

Typ	A[Integer]
Protokoll	

```
924 x ^ <Integer>
925 x: einInteger <Integer> ^ <Self>
```

definiert, wobei hier A[**Integer**] der (generische) Name des Typen ist. Diese Typdefinition wird jedoch nirgends hingeschrieben — sie ergibt sich immer neu aus der Instanziierung der parametrischen Typdefinition mit einem konkreten Typen. Es sind dann bei obiger Deklaration von a die Methodenaufrufe

```
926 a x: 12
927 a x + a x
```

zulässig,

```
928 a x: 'mal sehen, was passiert'
929 a x, a x
```

hingegen nicht. Für letztere wäre eine Typdeklaration

```
930 | a <A[String]> |
```

notwendig gewesen, die natürlich auch möglich ist.

Ein und dieselbe parametrische Typdefinition kann in einem Programm beliebig oft verwendet werden, selbst in derselben Deklaration:

mangelnde Zuweisungs- kompatibilität verschiedener Typinstanzen

```
931 | a <A[Integer]> b <A[String]> c <A[Boolean]> |
```



gibt **a**, **b** und **c** jeweils verschiedene Typen, die jedoch alle Instanzen der parametrischen Definition von **A** sind. Dennoch sind **a**, **b** und **c** wechselseitig nicht zuweisungskompatibel; sie haben tatsächlich verschiedene Typen. **a** ist jedoch mit **d** wie in

932 | **d <A[Integer]>** |

deklariert zuweisungskompatibel und umgekehrt, da beide denselben Typen haben.

29.2 Collections als Standardanwendungsfall für parametrischen Polymorphismus

Eine wichtige Gruppe von Klassen, die Sie in den letzten beiden Kurseinheiten kennengelernt haben, sind die sog. Collection-Klassen. Auch diese bilden jeweils einen Typ, so dass Variablen, die auf eine Collection verweisen, mit diesem Typ deklariert werden können.

Nun dienen Collections ja u. a. dem Zweck, $:n$ -Beziehungen zwischen einem Objekt und mehreren anderen zu ermöglichen, indem sie dafür Zwischenobjekte zur Verfügung stellen (s. Kurseinheit 2, Kapitel 13). Und so bilden die mit den Collection-Klassen assoziierten Typen auch nur die Typen für die Zwischenobjekte. Was man jedoch eigentlich bei der Deklaration von n -wertigen Attributen angeben (deklarieren) möchte, ist der Typ der in Beziehung stehenden Objekte.

Problemstellung

Wenn das Attribut beispielsweise **kinder** heißt und man damit eine Person mit einer Menge anderer Objekte vom Typ **Person**, den Kindern, in Beziehung setzen möchte, dann nutzt es nichts, wenn man **kinder** vom Typ **Person** deklariert — es könnte dann höchstens eine Person enthalten und nicht mehrere. Was man vielmehr gern hätte, wäre etwas, das dem Array-Typkonstruktor **array [<Bereich>] of <Elementtyp>** (spitze Klammern hier wieder als Begrenzer von metasyntaktischen Variablen) von PASCAL gleicht: Im gegebenen Beispiel würde man gern deklarieren, dass **kinder** den Typ **Collection of Person** haben soll. Genau das tut

Angabe des Elementtypen einer Collection

Klasse	Person
benannte Instanzvariablen	kinder <Collection[Person]>
Instanzmethoden	

933 ...

Passend dazu ist es möglich, **Collection** in STRONGTALK als parametrischen Typ wie folgt zu definieren:

parametrisch typisierte Collection

Typ	Collection
Typvariablen	E
Protokoll	

934 at: einIndex <Integer> ^ <E>



```
935 at: einIndex <Integer> put: einElement <E> ^ <Self>
```

Das Programmfragment

```
936 | p <Person> |
937 p := Person new.
938 p kinder at: 1 put: Person new.
939 p kinder at: 2 put: Person new.
940 (p kinder at: 1) kinder at: 1 put: Person new
```

ist demnach typkorrekt (und weist p zwei Kinder und ein Enkelkind zu).

Ein anderes Beispiel für eine parametrische Definition einer Collection ist **Dictionary**: Hier sollte nicht nur der Element-, sondern auch der Schlüsseltyp variabel gehalten werden. Eine entsprechende parametrische Typdefinition, diesmal mit zwei Typparametern, kann wie folgt aussehen:

<u>Typ</u>	Dictionary
<u>Typvariablen</u>	S E
<u>Supertyp</u>	Collection[E]
<u>Protokoll</u>	

```
941 at: einSchlüssel <S> ^ <E>
942 at: einSchlüssel <S> put: einElement <E> ^ <Self>
```

Dabei ist der parametrische Typ **Dictionary** ein Subtyp des ebenfalls parametrischen Typs **Collection**. Man beachte, dass der Typparameter E hier bereits in der Supertypdeklaration verwendet wird. Ein Dictionary, in dem Integer auf beliebige Objekte abgebildet werden, erhält man dann durch die Instanziierung **Dictionary[Integer, Object]**. Es ist mit einer Variable vom Typ **Collection[Object]** zuweisungskompatibel. Auf die Einzelheiten des Subtypings bei parametrischen Typen wird in Kapitel 30 eingegangen.

29.3 Parametrischer Polymorphismus und Inklusionspolymorphie

Nun war die Speicherung von Personen in Collections, wie sie oben benötigt wurde, auch schon ohne den parametrischen Polymorphismus möglich, nämlich per Inklusionspolymorphie (Subtyping). So würde es zunächst ausreichen, wenn **Collection** wie folgt definiert wäre:

<u>Typ</u>	Collection
<u>Protokoll</u>	

```
943 at: einIndex <Integer> ^ <Object>
944 at: einIndex <Integer> put: einObjekt <Object> ^ <Self>
```



An die Stelle der Typvariable **E** tritt also der (konkrete) Typ **Object**. Da in STRONGTALK alle Typen Subtypen von **Object** sind, kann man jedes beliebige Objekt in einer solchen Collection speichern. In der Klasse **Person**, die **Collection** verwendet, würde dann **kinder** schlicht als vom Typ **Collection** (ohne Typparameter) deklariert. Das obige Programmfragment (Zeilen 936–940) könnte dann auch beinahe so bleiben, bis auf eine kleine Ausnahme: Zeile 940 enthält jetzt einen Typfehler, da das Ergebnis von **p kinder at: 1** vom Typ **Object** ist und das Protokoll von **Object** keine Methode **kinder** unterstützt. Es wäre also erst noch eine Typumwandlung von **Object** nach **Person**, ein *Down cast* (s. Kapitel 27), vorausgesetzt. Deren Zulässigkeit ist aber davon abhängig, was wirklich in der Collection drinsteckt, und das kann der Compiler nicht (oder nur sehr aufwendig) feststellen. Die Lösung, die Inklusionspolymorphie bietet, beinhaltet also eine Sicherheitslücke in der statischen Typprüfung, die der parametrische Polymorphismus behebt.

Nun ist aber auch der parametrische Polymorphismus nicht ohne Makel. Zum einen wäre es ohne Inklusionspolymorphie nicht möglich, in einer Collection mit Elementtyp **XYZ** auch Objekte eines Subtyps von **XYZ** zu speichern. Solche *heterogenen Collections* kommen aber in der Praxis immer wieder vor, so dass man selbst bei Verwendung einer parametrischen Definition von Collections nicht auf Inklusionspolymorphie verzichten wird. Zum anderen wird die erhöhte Typsicherheit bei der Verwendung von parametrisch definierten Typen (wo man ja zumindest bei homogener, also ohne Ausnutzung der Inklusionspolymorphie, Belegung der mit einem Typparameter typisierten Variablen ohne Typumwandlungen auskommt) mit einer geringeren Typsicherheit innerhalb der Typdefinition (bzw. Klassendefinition) selbst erkauft. Dies verlangt nach Erklärung.

**Unzulänglichkeit des
einfachen
parametrischen
Polymorphismus**

Stellen Sie sich einen Collection-Typ **MyCollection** vor, dessen Werte solche Collections sein sollen, deren Elemente sortiert und summiert werden können. Dieser Typ sei ein Subtyp von **Collection** und verfüge weiterhin über entsprechende Methoden **sortieren** und **summieren**:

<u>Typ</u>	MyCollection
<u>Typvariablen</u>	E
<u>Supertyp</u>	Collection[E]
<u>Protokoll</u>	
945	sortieren ^ <Self>
946	summieren ^ <Number>

Intuitiv verlangt die Sortierbarkeit der Objekte vom Typ **MyCollection**, dass auf den Elementen eine Vergleichsfunktion definiert ist. Dies ist aber nicht für alle Typen und somit auch nicht für alle möglichen Belegungen der Typvariable **E** der Fall. Auch verlangt die Methode **summieren**, dass sich aus den Elementen einer solchen Collection ein Wert aggregieren lässt, der vom Typ **Number** oder einem Subtyp davon ist. Man kann daraus schließen,



dass die Elemente ebenfalls vom Typ **Number** sein oder zumindest Methoden besitzen müssen, die einen solchen Wert zurückliefern. Und so würde auch eine Implementierung der Methode **summieren** in etwa wie folgt aussehen:

```
947 summieren ^ <Number>
948   ^ elements
949     inject: 0
950     into: [ :summe <Number> :element <Number> | summe + element].
```

Das aber verlangt, dass der Elementtyp von **MyCollection Number** oder ein Subtyp davon sein muss, da sonst die Zuweisung an den formalen Blockparameter **element** nicht zulässig wäre. Insbesondere würde das Codefragment

```
951 | liste <MyCollection[String]> |
952 ...
953 liste summiere
```

zu einem Typfehler führen, weil in Zeile 950 einer Variable vom Typ **Number** ein Objekt vom Typ **String** zugewiesen wird. Nun kann aber die Definition des parametrischen Typs **MyCollection** nicht wissen, wie sie hinterher verwendet wird, und wenn eine Addition durchgeführt werden soll, ist sie darauf angewiesen, dass sie nur mit Typen von addierbaren Objekten instanziert wird. Es wird also die erhöhte Typsicherheit außerhalb der Typdefinition, nämlich bei ihrer Verwendung, durch eine verminderte Typsicherheit innerhalb erkauft.

Was man gerne hätte, um diesen Mangel zu beheben, wäre die Sicherheit, dass alle Typen, die für E eingesetzt werden können, bestimmte Eigenschaften haben, im gegebenen Beispiel, dass sie sortierbar und adäquat sind. Entsprechend sollte ein Typfehler nicht erst in Zeile 950 moniert werden, sondern bereits an der Stelle, an der die unzulässige Wertzuweisung an die Typvariable stattfindet, nämlich bei der Verwendung (der *Instanziierung*) der parametrischen Typdefinition in der Deklaration von Zeile 951. Genau das erlaubt der beschränkte parametrische Polymorphismus, der im nächsten Abschnitt behandelt wird. Zunächst jedoch noch zu einem anderen wichtigen Aspekt von parametrischem Polymorphismus und Subtyping.

mehr
Ausdrucksstärke benötigt

Unter den Typdefinitionen

Typ	A
-----	---

Typ	B
Supertyp	A

sowie

Typ	G
Typvariablen	T



```
954 x: <T> ^ <Self>
955 x ^ <T>
```

und den Variablen Deklarationen

```
956 | a <A> b <B> ga <G[A]> gb <G[B]> |
```

ist die Zuweisung

```
957 a := b
```

sicher zulässig. Nun könnte man annehmen, dasselbe sei auch für

```
958 ga := gb
```

der Fall. Dahinter verbirgt sich aber die Frage, ob $G[B]$ ein Subtyp von $G[A]$ ist, ob sich also die Subtypenbeziehung von B zu A auf entsprechende Typinstanzen vom selben parametrischen Typ überträgt. Intuitiv scheint dies der Fall, zumal beispielsweise

```
959 ga x: b
```

oder

```
960 ga x: gb x
```

aufgrund der Subtypenbeziehung von B zu A kein Problem darstellen sollte (aus Sicht der Typprüfung entspricht dies ja den Verhältnissen der Zuweisung aus Zeile 957). Nun ist aber nach den Regeln der statischen Typprüfung auch

```
961 ga x: a
```

erlaubt. Da ga aber nach der Zuweisung aus Zeile 958 lediglich ein Alias für das von gb bezeichnete Objekt ist, ga also auf ein Objekt vom Typ $G[B]$ verweist und dieser als Parametertyp von x : nur B zulässt, handelt es sich bei obigem Methodenaufruf um eine Typverletzung. Der Fehler liegt jedoch nicht im Methodenaufruf, der in der Tat typkorrekt ist, sondern vielmehr in der Zuweisung aus Zeile 958: $G[B]$ ist eben kein Subtyp von $G[A]$, nur weil B ein Subtyp von A ist (man beachte die Parallelität zu dem in Kapitel 25 beschriebenen Problem). Dieser Trugschluss ist einer der häufigsten Anfängerinnenfehler.

Subtypenbeziehung von tatsächlichen Typparametern überträgt sich nicht auf Instanzen parametrischer Typen

Selbsttestaufgabe 29.1

Prüfen Sie nach demselben Schema wie oben, ob $G[A]$ ein Subtyp von $G[B]$ sein darf.

Merken Sie sich also unbedingt, dass parametrischer Polymorphismus die Subtypenbeziehung seiner tatsächlichen Typparameter nicht auf die durch Instanziierung erzeugten Typen



überträgt. Dies wird auch in Abschnitt 29.5 noch eine besondere Rolle spielen. Vor diesem Hintergrund beinahe paradox erscheint, dass sich Subtyping jedoch dazu einsetzen lässt, das oben beschriebene Problem mit der „inneren Typsicherheit“ von parametrisch definierten Typen zu lösen.

29.4 Beschränkter parametrischer Polymorphismus

Obiges Beispiel hat gezeigt, dass die einfache Form des parametrischen Polymorphismus für Typsicherheit in der objektorientierten Programmierung nur teilweise nützlich ist: Da die Typvariablen selbst nicht typisiert sind, kann man innerhalb der Typdefinition (und der den Typ implementierenden Klassen) keine Aussagen über den Typ machen. Außerhalb, bei der Verwendung (Instanzierung) der Typdefinition, geht das schon, da hier die Typvariable durch einen Typ ersetzt ist.

Was man also gern hätte, ist, dass die Typvariable innerhalb der mit ihr parametrisierten Typdefinition selbst wertbeschränkt ist, und zwar derart, dass man bei den als Werte zulässigen Typen ein bestimmtes, benötigtes Protokoll voraussetzen kann. Die tatsächlichen Typparameter sind dann nicht mehr beliebig zu wählen, sondern nur noch aus solchen Typen, die die Einschränkungen erfüllen. Eine Möglichkeit, das zu erzielen, wäre, Metatypen einzuführen, deren Wertebereiche Typen mit durch die Metatypen vorgegebenen Eigenschaften sind. Diese Möglichkeit wird jedoch in der Praxis nicht genutzt.

wertbeschränkte
Typvariablen

Stattdessen verwendet man eine Art der Beschränkung des Wertebereichs von Typvariablen, die auf Subtyping beruht. Wenn man nämlich erzwingen kann, dass ein tatsächlicher Typparameter (also der Wert der Typvariable) Subtyp eines bestimmten Typs ist, der die benötigten Eigenschaften (Methoden) umfasst, dann ist damit alles erreicht, was man benötigt: Aufgrund der Regeln des Subtyping hat jeder solche Typ die Eigenschaften des Supertyps (s. Kapitel 26).

Supertypen als
Schranken

Ein solchermaßen durch einen Supertyp beschränkte parametrische Typdefinition ist die folgende:

Beispiel

Typ	MyCollection
Typvariablen	E < Number

Der Rest der Definition geht wie oben. Der Ausdruck `E < Number` im Abschnitt „Typvariablen“ ist Deklaration und Beschränkung zugleich; die Beschränkung ist aber wie gesagt keine Typisierung wie in normalen Variablen-deklarationen. Sie drückt vielmehr aus, dass die Typen, die als Werte für E eingesetzt werden dürfen, Subtypen von `Number` sein müssen. Die Deklaration aus Zeile 951 wird damit unzulässig und führt zu einem entsprechenden Typfehler während der statischen Typprüfung; die Deklaration



ist hingegen OK.

29.5 Rekursiv beschränkter parametrischer Polymorphismus

Rekursive Typen sind Typen, die sich in ihrer Definition selbst referenzieren. Ein Beispiel für einen rekursiven Typ hatten Sie oben schon kennengelernt: Der (zur Klasse **Person** gehörende) Typ **Person** hat Methoden, die **Person** als Parameter- bzw. Rückgabetypen haben. Rekursive Typen sind ein wichtiges Instrument der Programmierung: Ohne sie wären dynamische Strukturen wie beispielsweise verzeigerte Listen oder Bäume kaum möglich. Rekursive Typen machen aber auch bestimmte Probleme — so ist beispielsweise die strukturelle Äquivalenz zweier rekursiver Typen nicht so leicht festzustellen, da die dazu notwendige *Expansion rekursiver Typen* (also das Einsetzen der Struktur für jeden darin vorkommenden Typnamen; vgl. Abschnitt 22.1) unendlich große Definitionen ergibt.

Es ergibt sich nun ein weiteres Problem, wenn man in einer parametrischen Typdefinition den Typ eines Methodenarguments (eines formalen Parameters einer Methode) variabel halten möchte, dieser Typ aber ausgerechnet der definierte ist (eine *binäre Methode*; vgl. Fußnote 21 in Kapitel 4.3). So möchte man beispielsweise den Test auf Gleichheit so definieren, dass das Objekt, das gleich sein soll, vom selben Typ sein muss wie das, mit dem man Gleichheit feststellen möchte. Für den Typ **Object** schreibt man dazu einfach

parametrische
polymorphe
Definition binärer
Methoden

Typ	Object
Protokoll	
963 = einObjekt <Object> ^ <Boolean>	
964 ...	

für den Typ **Number**

Typ	Number
Protokoll	
965 = eineZahl <Number> ^ <Boolean>	
966 ...	

usw. Nun ist aber **Number** ein Subtyp von **Object**, so dass man die Deklaration von = eigentlich aus **Object** übernehmen könnte — wenn der Typ des Parameters automatisch so angepasst würde, dass er dem definierten Typ entspricht. In einem ersten Ansatz wäre man vielleicht versucht, den Gleichheitstest in **Object** einfach als = **einObject <Self> ^ <Boolean>** zu deklarieren, aber das würde, wenn die *Pseudo-Typvariable Self* beim Subtyping jeweils den Subtyp annehmen soll, zu einer kovarianten Redefinition mit den



bereits bekannten Problemen führen.⁵⁵ Auch hier bietet parametrischer Polymorphismus eine Alternative, wenn auch nicht ganz so, wie vielleicht erwartet.

Man ersetzt dazu zunächst den Typ des Parameters durch eine Typvariable T. Nun kann man schlecht

Lösung durch
Rekursion

Typ	T
Typvariablen	T
Protokoll	
967	= einT <T> ^ <Boolean>

schreiben, da der Typ dann keinen Namen hätte und somit auch nicht verwendbar (referenzierbar) wäre. Was man aber sehr wohl machen kann, ist, einen allgemeinen parametrischen Typ zu definieren, der nur dem Zweck des Gleichheitstests dient und der den Parametertyp des Tests variabel hält, wie in

Typ	Equatable
Typvariablen	T
Protokoll	
968	= einT <T> ^ <Boolean>

Man kann dann die gewünschte Rekursion indirekt, nämlich per Definition eines nicht parametrischen Typs als Subtyp des parametrisierten Typs **Equatable** herstellen, wobei man den zu definierenden Typ gleichzeitig als tatsächlichen Typparameter einsetzt. So liefert z. B.

Typ	Integer
Supertyp	Equatable[Integer]

eine Methode mit der Signatur = `einT <Integer> ^ <Boolean>` im Protokoll von **Integer**. Allerdings kann man so nicht erzwingen, dass bei der Definition des Typs **Integer** oben genau **Integer** als tatsächlicher Typparameter eingesetzt wird; es hätte auch jeder andere Typ, z. B. **String**, sein können — der Gleichheitstest wäre dann mit = `einT <String> ^ <Boolean>` falsch deklariert.

Genau diese Beschränkung des tatsächlichen Typparameters kann man nun mit einer stilistischen Figur erreichen, die vermutlich manch einer von Ihnen erhebliche Kopfschmerzen bereiten wird (zumindest macht sie das mir immer wieder aufs neue): Man beschränkt den formalen Typparameter T von **Equatable** auf einen Subtyp von **Equatable[T]**, wobei das Vorkommen von T in **Equatable[T]** eine Verwendung der gerade erst eingeführten Typvariable T darstellt.

⁵⁵ Genau das macht übrigens auch EIFFELS like Current (s. Abschnitt 52.5.2).



Typ

Equatable

Typvariablen

T < Equatable[T]

Protokoll

969 = einT <T> ^ <Boolean>

verlangt also im obigen Beispiel der Typdefinition von `Integer` als Subtyp einer Instanziierung der parametrischen Definition von `Equatable`, dass der tatsächliche Typparameter `Integer` ein Subtyp von `Equatable[Integer]` sein muss. Genau das sagt aber die obige Typdefinition von `Integer` aus! Stünde dort `Equatable[String]` oder irgend etwas anderes als Typschranke, wäre dies nicht mehr der Fall (s. Abschnitt 29.3) und die Definition von `Integer` verursachte einen statisch feststellbaren Typfehler.

Wenn Sie hier ein Verständnisproblem haben, trösten Sie sich — es dauert eine Weile, bis man es verstanden hat, und noch länger, bis solche Figuren zum aktiven Repertoire gehören. Gleichwohl sollten Sie sich damit befassen: Das JAVA-Collections-Framework in der Version von JAVA 5 ist voll solcher Typdefinitionen, nicht, weil sie schön sind, sondern weil man sie braucht, um das Framework typsicher zu machen, ohne seine Flexibilität zu opfern. Auch Sie werden, wenn Sie objektorientiert programmieren, über kurz oder lang solche Konstrukte von sich geben müssen.

ein paar Worte zum
Trost

30 Parametrischer Polymorphismus und das Kovarianzproblem

In gewisser Weise hat man es beim rekursiv beschränkten parametrischen Polymorphismus wie oben vorgestellt mit einem Fall von kovarianter Redefinition zu tun: Der Parametertyp der Methode `=` ändert sich mit dem Empfängertyp. Allerdings ergibt sich daraus, anders als bei der Verwendung von `Self als Typvariable`, kein Widerspruch zur Kontravarianzregel des Subtyping, denn `Integer` wird dadurch unmittelbar ja lediglich zu einem Subtyp von `Equatable[Integer]` und nicht etwa von `Equatable[Object]`. Tatsächlich sind `Equatable[Integer]` und `Equatable[Object]` ja zwei vollkommen verschiedene Typen (mit disjunkten Wertebereichen) und `Equatable[T]` ist gar kein Typ (so dass man auch keine Variable mit ihm deklarieren kann), so dass keinerlei Zuweisungskompatibilität und damit auch kein Problem mit Typkorrektheit besteht.

Trotzdem stellt sich die Frage, ob sich das in Abschnitt 26.3 angesprochene allgemeine Problem der wünschenswerten kovarianten Redefinition von Eingabeparametern in Methoden mittels parametrischen Polymorphismus nicht irgendwie lösen lässt. Die Antwort ist unbefriedigend: nur zum Teil.

So kann man, um das Beispiel von Dokumenten und Drucken aus Abschnitt 26.3 wieder aufzugreifen, einen parametrischen Typ `Dokument` wie folgt definieren:



Typ

Dokument

Typvariablen

T < Drucker

Protokoll

970 druckenAuf: einDrucker <T> ^ <Self>

Die Deklaration von **Zeichnung** mit Typparameter T als Subtyp von **Dokument** vorausgesetzt, lassen sich die folgenden Variablen-deklarationen bilden:

971 | z <Zeichnung[Plotter]> p <Plotter> l <Zeilendrucker> |

Weiterhin die Deklarationen von **Plotter** und **Zeilendrucker** als Subtypen von **Drucker** vorausgesetzt wäre ein Methodenaufruf

972 z druckenAuf: p

typkorrekt,

973 z druckenAuf: l

hingegen nicht. Allerdings ist die Assoziation von **Zeichnung** mit **Plotter**, die Kovarianz, in keiner Typdefinition festgehalten, sondern lediglich in der Deklaration von z. Es hindert einen insbesondere nichts daran, dieselbe oder eine andere Variable als vom Typ **Zeichnung[Zeilendrucker]** zu deklarieren. Man beachte, dass es anders als im obigen Beispiel von **Equatable**, wo ja der Typparameter auf den definierten Typ selbst eingeschränkt wurde, hier keine Möglichkeit gibt, einen bestimmten Wert für einen Typparameter vorzuschreiben.

Was man allerdings tun könnte, ist, **Zeichnung** als Subtyp von **Dokument[Plotter]** zu definieren. Dies hat jedoch den Nachteil, dass **Zeichnung** damit kein Subtyp mehr von **Dokument** und, wie auch zuvor schon **Zeichnung [Plotter]** kein Subtyp von **Dokument[Drucker]** ist (s. Abschnitt 29.3), wodurch die Zuweisungskompatibilität mit entsprechend deklarierten Variablen verlorengeht. Kovariante Redefinition bei gleichzeitiger Inklusionspolymorphie lässt sich auch mittels parametrischer Typen nicht hinbekommen.

31 Grenzen der Typisierung

Wie Sie sehen, ist das Problem der kovarianten Redefinition ziemlich hartnäckig. Man muss aber gar nicht so weit gehen, um an die praktischen Grenzen der Typisierung zu gelangen: Bereits der Ausdruck

974 x reciprocal

beinhaltet einen Typfehler, wenn nicht sichergestellt ist, dass x nicht 0 enthält. Nun könnte man einen Typ **NotZero** definieren und x als von diesem Typ deklarieren, womit der obige



Ausdruck kein Problem mehr wäre; mit den hier vorgestellten Mitteln der statischen Typprüfung wäre dann aber schon die einfache Zuweisung

975 `x := y - z`

nicht mehr auf Zulässigkeit prüfbar. Selbst wenn es Typsysteme gibt, die das können⁵⁶, so sind diese kaum praxistauglich.

32 Fazit

Typsysteme sind immer noch Gegenstand aktiver Forschung. Während die prozeduralen und objektorientierten Programmiersprachen eher pragmatisch an das Thema herangehen, sind auf dem Gebiet der funktionalen Programmiersprachen ausfeilte Theorien entwickelt worden, die nach und nach auf andere Programmiersprachen übertragen werden. Die meisten der heute in Gebrauch befindlichen objektorientierten Programmiersprachen sind hingegen nicht die Quintessenz umfangreicher theoretischer Überlegungen, sondern vielmehr das Produkt von Ideen, Experimenten und einer ganzen Menge praktischer Zwänge.

So ist denn die Darstellung von Typsystemen in dieser Kurseinheit eher dazu gedacht, an einen gewissen Konsens bei der Typisierung objektorientierter Programmiersprachen heranzuführen denn das Thema theoretisch aufzubereiten. Eine gute, nicht allzu theorielastige Einführung in Typsysteme für die objektorientierte Programmierung ist das Buch „Object-Oriented Type Systems“ von JENS PALSBERG und MICHAEL SCHWARTZBACH; es ist recht dünn und dabei noch einigermaßen angenehm zu lesen. Sehr viel weitergehend ist das Buch „Types and Programming Languages“ von BENJAMIN PIERCE, das einen Standard darstellt.



33 Lösungen zu den Selbsttestaufgaben

Selbsttestaufgabe 20.1 (Seite 156)

Typ Boolean

Protokoll

976 `and: aBoolean <Boolean> ^ <Boolean>`
977 `or: aBoolean <Boolean> ^ <Boolean>`
978 `not ^ <Boolean>`
979 `...`

⁵⁶ Tatsächlich gibt es Typsysteme, die selbst Turing-äquivalent sind, mit denen man also alle Prüfungen durchführen kann, die auch vom Programm selbst durchgeführt werden könnten.



Selbsttestaufgabe 26.1 (Seite 172)

Man definiert einfach die Addition als Methode des Typs **Punkt** und redefiniert (überschreibt) die Methode in **DreiDPunkt** entsprechend. Allerdings muss man dann noch sagen, was passieren soll, wenn zu einer Instanz vom Typ **DreiDPunkt** (als Empfängerobjekt) eine Instanz vom Typ **Punkt** addiert werden soll; die Regeln der Zuweisungskompatibilität verbieten das ja nicht! (Eine mögliche Lösung wäre *Multi dispatch*.)

Selbsttestaufgabe 29.1 (Seite 191)

Wenn dem so wäre, dann wäre

980 **gb** := **ga**

zulässig. Der Ausdruck

981 **gb** **x**

der den deklarierten Typ **B** hat, würde dann aber (wegen des Alias von **gb** auf das Objekt in **ga**) ein Objekt vom Typ **A**, also einem Supertyp, zurückliefern. Also ist auch **G[A]** kein Subtyp von **G[B]**!



Kurseinheit 4: JAVA

Da die meisten unter Ihnen vermutlich nie in die Verlegenheit kommen werden, beruflich in SMALLTALK zu programmieren, wenden wir uns in dieser und der nächsten Kurseinheit anderen objektorientierten Programmiersprachen zu. Am meisten Raum nimmt dabei JAVA ein, zum einen, weil es immer noch die am weitesten verbreitete objektorientierte Programmiersprache ist, zum anderen, weil es in der Darstellung viele Parallelen zu SMALLTALK zu ziehen erlaubt und damit eine gute Brücke zu weiteren Programmiersprachen schlägt. Unter theoretischen (und damit auch akademischen) Gesichtspunkten nimmt JAVA allerdings keine besondere Stellung ein und für seinen Erfolg war vermutlich eher das Internet (JAVA spielte vormals die Rolle, die heute JAVASCRIPT spielt) als ein besonders genialer Sprachentwurf verantwortlich.

Link



Aus der Distanz betrachtet ist JAVA eine Mischung aus SMALLTALK und C++. Von C++ wurden weite Teile der Syntax und der statischen Typprüfung übernommen sowie der eher klassisch prozedurale Charakter (Methodenaufrufe anstelle von Nachrichtenversand), von SMALLTALK die weitgehende Objektorientierung (es gibt keinen Code außerhalb von Klassen), die *Einfachvererbung* sowie die *Garbage collection* (um nur die wichtigsten zu nennen). Wenn man allerdings genauer wissen will, was JAVA ist und warum es so ist, wie es ist, dann muss man einen Blick in seine Geschichte werfen. Interessierten empfehle ich unbedingt, „The Long Strange Trip to Java“ von PATRICK NAUGHTON zu lesen; danach wird Ihnen vermutlich so einiges klar.

**JAVA als Kind von
SMALLTALK und C++**

Link



34 Das Programmiermodell JAVAs

JAVA ist in vielerlei Hinsicht (und vor allem im Vergleich zu SMALLTALK) eine konventionelle Programmiersprache. Programme werden als Quelltext in sog. *Compilation units* gespeichert, die gewöhnlich Dateien sind und die — jede für sich — immer als Ganzes übersetzt werden.⁵⁷ Das Ergebnis der Übersetzung ist jedoch kein direkt ausführbarer Maschinencode, sondern ein sog. *Bytecode*, der von einer virtuellen Maschine, der *Java Virtual Machine* (JVM), interpretiert werden muss. Das sonst übliche Linken der einzelnen Klassen (genauer: des zu den Klassen gehörenden Bytecodes, die sog. Class files) wird dabei durch das Class

⁵⁷ Die Entwicklungsumgebung VISUALAGE FOR JAVA, die Vorgängerin von ECLIPSE, machte hier eine Ausnahme und sah — wie Smalltalk — die Speicherung von Klassen in einem Image vor. Eclipse hat übrigens bis heute eine Java Browsing Perspective, die dem System Browser von Smalltalk ähnelt.



loading der JVM ersetzt. Vorteile des ganzen sind eine größere Flexibilität bei der Entwicklung und Verteilung von Anwendungen sowie eine weitgehende Plattformunabhängigkeit: JAVA-Programme können, soweit sie nicht von bestimmten Eigenheiten der Betriebssysteme abhängen (man denke z. B. an die unterschiedliche Handhabung von Groß-/Kleinschreibung — JAVA ist case sensitive, Windows nicht!), auf jedem Rechner und Betriebssystem laufen, für die es eine JVM gibt.

Während JAVA als Programmiersprache anfangs noch recht klein und überschaubar war (zumindest im Vergleich zu C++, einem ihrer Hauptkonkurrenten), so ist die Sprachdefinition heute ein Moloch. Mit dem wachsenden Nutzerinnenkreis sind auch die Anforderungen an die Sprache gewachsen, und mit dem JAVA Community Process wurde aktiven Entwicklerinnen die Möglichkeit eingeräumt, Vorschläge zur Spracherweiterung zu machen. Dabei ist jedoch — aufgrund der mittlerweile riesigen Menge an Software, die in JAVA geschrieben ist — stets auf Rückwärtskompatibilität zu achten, so dass revolutionäre Verbesserungen kaum möglich sind; stattdessen wird hinzugefügt. Das unterliegende Programmiermodell ist so immer dasselbe geblieben — und wird es wohl auch immer bleiben.

rückwärtskompatible Weiterentwicklung

Die grundlegenden Werkzeuge der JAVA-Programmierung sind neben dem Editor der JAVA-Compiler `javac`, die JVM, die JAVA-Klassenbibliothek (das sog. *Application Programming Interface*, API) und natürlich die Dokumentation (API-Dokumentation und Sprachdefinition). Gerade für die JAVA-Programmierung gibt es jedoch zahlreiche *integrierte Entwicklungsumgebungen* (IDEs) und es ist niemandem zu raten, diese Werkzeuge links liegen zu lassen — die einmal einen Teil ihrer wertvollen Lebenszeit mit dem richtigen Setzen des sog. Class path verbracht hat, wissen, wovon ich rede. Auf der anderen Seite sind diese IDEs sehr komplex und erschlagen gerade Anfängerinnen mit ihrem Funktionsumfang. Das ist auch der Grund, warum in diesem Kurs keine JAVA-Installation von Ihnen verlangt wird — wer es aber wissen und wer mit JAVA experimentieren möchte, die will ich keinesfalls davon abhalten.

wichtige Werkzeuge der JAVA- Programmierung

Um ein JAVA-Programm, bestehend aus einer Menge von Class files, auszuführen, muss man eine Klasse angeben, die eine Startmethode besitzt. Diese Startmethode heißt immer gleich; ihre Signatur hat die leicht zu merkende und immer wieder gern eingetippte Form

Ausführung eines JAVA-Programms

982 `public static void main(String[] args)`

Dabei ist `main` der Name der Methode; was die anderen Elemente bedeuten, werden Sie im Laufe dieser Kurseinheit noch lernen. Auf Betriebssystemebene übergibt man dann einfach der JVM bei ihrem Aufruf den Namen der Klasse als Parameter, beispielsweise mit

983 `java MeineTollsteKlasseBisher`

Klassen werden in JAVA wie in SMALLTALK per Konvention immer großgeschrieben.



Art des Deployment, also wie in JAVA programmierte Anwendungen in die Anwendung gehen. Während früher alle namhaften Web-Browser (per Plug-in) den Start von in Webseiten eingebetteten JAVA-Anwendungen erlaubten, bleibt heute praktisch nur noch die Verteilung per sog. JAVA Archive (einer .jar-Datei). Zu deren Ausführung ist jedoch eine Installation der JVM vonnöten.

35 Objekte und Typen

JAVA-Programme sind objektorientierte Programme — zur Laufzeit bestehen sie aus einer Menge interagierender Objekte. Dabei ist JAVA stark typisiert: Jedes Objekt gehört zum Wertebereich eines oder mehrerer Typen. Anders als in SMALLTALK gibt es in JAVA neben Objekten auch Werte wie Zahlen, Zeichen und Wahrheitswerte, die keine Objekte sind.

In JAVA werden zunächst sechs Arten von Typen unterschieden: *primitive Typen*, *Klassentypen*, *Interfacetypen*, *Array-Typen*, *Aufzählungstypen* und *Annotationstypen*. Von Klassen- und Interfacetypen gibt es seit JAVA 5 auch parametrisierte Varianten, die denen STRONGTALKS stark ähneln (die JAVA-Genericst stammen teilweise von Autoren STRONGTALKS). Die primitiven Typen sind mit der Sprachdefinition festgelegt; es sind dies `byte`, `short`, `int`, `long`, `float`, `double`, `boolean` und `char`. Sie unterscheiden sich nicht sonderlich von denen gängiger, statisch typgeprüfter prozeduraler Sprachen. Alle anderen Typen werden durch Typkonstruktion mittels entsprechender Sprachkonstrukte in JAVA selbst definiert. Ein Teil dieser Typen ist jedoch mit der Sprachdefinition fest vorgegeben: So sieht JAVA für jeden primitiven Typ einen im wesentlichen gleichnamigen (mit Ausnahme von `int` und `char` bis auf Großschreibung) Referenztyp vor, dessen Werte jeweils einen Wert eines entsprechenden primitiven Typs aufnehmen können. Diese Typen, namentlich `Byte`, `Short`, `Integer`, `Long`, `Float`, `Double`, `Boolean` und `Character`, nennt man deswegen *Wrapper-Typen* (sie wickeln gewissermaßen einen Wert eines primitiven Typs ein). Weitere Typen, die die JAVA-Sprachdefinition voraussetzt, sind `Object`, `String`, `Throwable`, `Error`, `Exception`, `RuntimeException` und `Thread`. Diese werden an entsprechenden Stellen unten weiter erläutert.

unterschiedliche Arten von Typen

Genau wie in SMALLTALK müssen auch in JAVA die Objekte irgendwo herkommen. Neben der Instanziierung von Klassen, wie Sie sie auch schon von SMALLTALK her kennen, gibt es dafür auch in JAVA Literale.

35.1 Literale

In JAVA gibt es Literale für Zahlen, Zeichen und Strings. Für Arrays gibt es, da sie nicht aus Literalen zusammengesetzt sein müssen, etwas ähnliches, nämlich die sog. *Array-Initialisierer*; sie werden in Kapitel 41 behandelt. Ob `true` und `false` bzw. `null` in JAVA Literale



oder Schlüsselwörter sind, hängt vom Standpunkt ab: Die meisten Syntaxeditoren behandeln sie wie Schlüsselwörter, gemäß Sprachdefinition handelt es sich aber um Literale, die die beiden Booleschen Wahrheitswerte „wahr“ und „falsch“ bzw. den Wert des Nulltypen (`UndefinedObject` in SMALLTALK, in JAVA unbenannt) repräsentieren.

Zahlenliterale in JAVA können eine Vielzahl von Formen annehmen. Für drei besondere Werte im Fließkommabereich stehen allerdings keine Literale, sondern nur Konstanten zur Verfügung: Es sind dies `NaN` (für das englische Not a Number) sowie `POSITIVE_INFINITY` und `NEGATIVE_INFINITY`. Zeichenliterale werden in JAVA durch einfache Anführungsstriche eingeschlossen, String-Literale durch doppelte. Beide können Escape-Sequenzen zur Darstellung von Sonderzeichen enthalten.

Syntax



WIKIPEDIA

Anders als in SMALLTALK sind in JAVA Klassen keine Objekte. Dennoch muss man in Programmen gelegentlich Klassen als Werte angeben. Das geht dann nicht (wie in SMALLTALK) per Angabe des Klassennamens (der dort ja zugleich als Pseudovariable definiert war), sondern mittels eines sog. **Klassenliterals**. Dies besteht aus dem Namen der Klasse, gefolgt von `.class`, also beispielsweise

Klassenliterale

984 `Object.class`

Der Typ eines solchen Klassenliterals ist `Class<T>`, also im obigen Beispiel `Class<Object>`. Da in JAVA-Programmen Klassennamen auch direkt vorkommen dürfen, so z. B. als Empfänger beim Aufruf von Klassenmethoden oder in Typtests wie `instanceof` (s. Abschnitt 35.4), sind Klassenliterale eher selten anzutreffen (die ganze Unterscheidung von Klassennamen und Klassenliteralen in JAVA ist m. E. alles andere als glücklich).

35.2 Gleichheit und Identität

In JAVA wird genau wie in SMALLTALK zwischen Gleichheit und Identität von Objekten unterschieden. Die Gleichheit von Objekten wird mittels der Methode `equals(.)` (wobei `(.)` hier für einen nicht näher spezifizierten Parameter steht), die Identität mittels `==` (bzw. `!=` für das Negat) geprüft. `equals(.)` wird von der Klasse `Object` (in JAVA genau wie in SMALLTALK die Superklasse aller Klassen) geerbt und sollte in den Subklassen der jeweiligen Bedeutung von Gleichheit entsprechend überschrieben werden. Die Verwechselung von `equals(.)` und `==` ist auch in JAVA ein ziemlich häufiger Programmierfehler (vgl. Abschnitt 1.4 in Kurseinheit 1). Das Gleichheitszeichen `=` steht in JAVA übrigens (genau wie in C, C++ und C#) für die Wertzuweisung, was ich persönlich für eine der großen Tragödien der Informatik halte.

35.3 Variablen und Zuweisungen

JAVA ist eine stark typisierte Sprache: Alle Ausdrücke haben einen Typ. Das gilt auch für Variablen, deren Typ bei ihrer Deklaration angegeben werden muss.



In JAVA gibt es Variablen mit *Referenz-* und mit *Wertsemantik*. Welche Semantik eine Variable hat, richtet sich nach ihrem Typ. Typen, die zu Variablen mit Wertsemantik führen, sind die oben genannten *primitiven*, namentlich **byte**, **short**, **int**, **long**, **float**, **double**, **boolean** und **char**. Variablen, die mit einem anderen Typ deklariert werden, haben Referenzsemantik.

Variablen werden in JAVA genau wie in SMALLTALK explizit per Zuweisungsoperator **=** und implizit per Methodenaufruf Werte zugewiesen. Der Inhalt von Variablen primitiver Typen ist ein entsprechender Wert, der der anderen Typen immer eine Referenz auf ein Objekt.

Es gibt in JAVA genau wie in SMALLTALK keine Möglichkeit, Pointervariablen explizit zu deklarieren; es gibt also insbesondere beim Methodenaufruf (und den damit verbundenen impliziten Zuweisungen) auch in JAVA kein *Call by reference*, sondern nur ein *Call by value*. Daran ändert auch nichts, dass Variablen, die mit Referenztypen deklariert sind, Referenzsemantik haben — bei den impliziten Zuweisungen eines Methodenaufrufs wird den formalen Parametern immer eine Kopie des Zeigers übergeben. Siehe dazu auch die Bemerkungen in Abschnitt 4.3.2 (Kurseinheit 1) und Kapitel 37.

**Wertzuweisungen
und Inhalt von
Variablen**

Seit JAVA 5 können Werte primitiven Typs an Variablen der entsprechenden Wrapper-Typen direkt zugewiesen werden und umgekehrt; dies nennt man **Auto boxing** bzw. **Auto unboxing**. Dabei können allerdings, genau wie bei der Handhabung bestimmter Werte als Objekte in SMALLTALK (s. Abschnitt 1.4 in Kurseinheit 1), unerwartete Phänomene auftreten: Zwei Objekte, die den gleichen Wert repräsentieren, sind zwar immer gleich, müssen aber nicht identisch sein. Man muss also auch in JAVA genau überlegen, ob man die Equals-Methode oder den Test auf Identität (**==**) verwendet; liegt man daneben, handelt man sich schwer zu findende Programmierfehler ein.

**Auto boxing und
unboxing**

35.4 Operationen und Methoden

Einhergehend mit der Trennung zwischen primitiven und Referenztypen gibt es in JAVA auch eine Trennung zwischen Operationen und Methoden: Auf den primitiven Typen sind mit der Sprachdefinition bestimmte Operationen (wie die mathematischen und die Booleschen) fest vorgegeben; alle anderen Operationen müssen als Methoden in Klassen definiert werden. Da primitive Typen aber keine Klassen sind, gibt es z. B. für mathematische Funktionen eine spezielle Klasse **Math**, in der Funktionen wie **sin(.)** als Klassenmethoden hinterlegt sind. Umgekehrt ist es den Klassendefinitionen nicht erlaubt, Operatoren (wie die binären Methoden in SMALLTALK; s. Abschnitt 4.1.2 in Kurseinheit 1) zu definieren. Die Trennung von primitiven und Referenztypen ist also konsequent durchgezogen⁵⁸, was JAVAs Charakter als Programmiersprache anfangs

**Trennung zwischen
Operationen und
Methoden**

⁵⁸ Ausnahmen: Der Additionsoperator **+** ist auch für Strings definiert und bedeutet dort die Konkatenation (Aneinanderfügung). Strings sind aber sowieso ein Sonderfall in JAVA. Außerdem sind Identitätstests (**==** und **!=**) und Zuweisung (**=**) Operatoren, die für alle Typen definiert sind.



wesentlich geprägt und was ihr bei manchen den Ruf einer hybriden Programmiersprache (halb objektorientiert, halb imperativ) eingebracht hat. Aus meiner Sicht ist die Unterscheidung zwischen Objekten und Werten, wie sie in JAVA vorgenommen wurde, aber sinnvoll: Werte haben weder Identität noch Zustand, was also macht sie zu Objekten?

Die Methoden JAVAs kann man in Prozeduren und Funktionen aufteilen, wobei der einzige Unterschied ist, dass Prozeduren nichts zurückgeben und deswegen **void** anstelle des Rückgabetyps deklarieren. Es ist dies eine der vielen Erbschaften von der Programmiersprache C.

Void-Methoden als Prozeduren

Ein Operator, der speziell für Referenztypen zur Verfügung steht, ist der Typtest **instanceof**: Er erlaubt es zu prüfen, ob ein Objekt Element (Instanz) eines Typs ist. Dabei wird nicht zwischen *direkten* und *indirekten Instanzen* unterschieden: **x instanceof Object** wertet also immer zu **true** aus, egal, für welches Objekt **x** steht. Mehr zur Bedeutung des Typtests in Kapitel 44.

Typtestoperator instanceof

35.5 Zuweisungskompatibilität

In JAVA ist die Typkonformität und damit die Zuweisungskompatibilität unter Referenztypen an Subtyping gebunden: Damit Ausdrücke von einem Typ Variablen eines anderen zugewiesen werden können, müssen die Typen entweder gleich sein oder es muss eine Subtypbeziehung zwischen den beiden bestehen. In den meisten Fällen muss die Subtypbeziehung explizit deklariert werden; JAVA setzt also auf *nominale Typkonformität*. Dies hat den in Abschnitt 22.2 von Kurseinheit 3 beschriebenen Vorteil der Filterfunktion, aber auch den Nachteil, dass Subtypen ihre Supertypen namentlich kennen müssen, was insbesondere bei verteilten Anwendungen, deren Teile nicht von vornherein füreinander entworfen waren (Web Services beispielsweise), von Nachteil ist. Weiterhin verlangt das Subtyping JAVAs, dass die Typen geerbter Variablen (Felder und Parametertypen von Methoden) nicht abgeändert werden — JAVA verlangt also *Novarianz*. Damit wird zumindest eine statisch-semantische Substituierbarkeit von Objekten der Subtypen gegen die ihrer Supertypen sichergestellt (vgl. Abschnitt 26.3 und Kapitel 54 in Kurseinheit 6). Allerdings kann der Rückgabetyp von Methoden kovariant redefiniert werden; mehr dazu in Abschnitt 36.4.

Im Gegensatz zu ihren Vorläufern SMALLTALK und C++ ist JAVA eine Sprache mit einem strengen Typsystem. Das heißt insbesondere, dass in JAVA *alle* Typfehler entweder schon während der Übersetzung vom Compiler oder aber während der Ausführung vom Laufzeitsystem, dann aber schon zum frühestmöglichen Zeitpunkt, nämlich bei einer Wertzuweisung (bei der ja die Verletzung einer Typinvariante entsteht), abgefangen werden. In einem Fall kann man jedoch der Meinung sein, dass die Typinvarianten JAVAs zu lax gefasst sind, also Zuweisungen gestattet werden, von denen man nicht ausschließen kann, dass sie in der Folge zu einem Typfehler führen. Mehr dazu in Kapitel 41.

Auftreten von Typfehlern



36 Klassen

JAVA ist (wie SMALLTALK und alle in der nächsten Kurseinheit behandelten Sprachen) klassen- und nicht prototypenbasiert; man programmiert also, indem man Klassendefinitionen angibt. Wie bereits in Kapitel 34 erwähnt gibt es in JAVA keine Anweisungen außerhalb von Klassen (sieht man einmal von Import-Anweisungen ab).

Wie dort ebenfalls bereits erwähnt gibt es in JAVA einen relativ engen

Klassen und Dateien

Zusammenhang zwischen Klassen und Dateien: Jede Datei enthält die Definition einer Klasse, die den Dateinamen (ohne Erweiterung) als Namen trägt. Eine Datei (oder *Compilation unit*) kann auch mehrere Klassen enthalten, die dann natürlich anders heißen — unter ihnen darf jedoch keine als **public** deklariert werden. Schon weil die meisten integrierten Entwicklungsumgebungen für JAVA heute auf Dateibasis arbeiten und Dateien die Basis von vielen Versionsverwaltungssystemen sind, ist es jedoch wenig gebräuchlich, mehrere Klassen in einer Datei zu definieren (es sei denn, es handelt sich um *innere Klassen*; s. u.).

Nun hatten wir ja bereits in Kurseinheit 3, Kapitel 28, gesagt, dass Klassen

Klassen und Typen

und Typen zunächst zwei verschiedene Konzepte sind, dass aber eine Typdefinition aus einer Klassendefinition abgeleitet werden kann. Genau so verhält es sich in JAVA: Jede Klasse definiert ihren eigenen Typ. Genauer: Jede Klasse spezifiziert einen Typ, der genauso heißt wie die Klasse und der als Eigenschaften die Felddefinitionen und die Methodendeklarationen der Klasse enthält. Eine Deklaration

985 <Klassenname> <variablenname>

ist also eine gültige Variablen-deklaration in JAVA. Man beachte, dass, anders als in STRONG-TALK, in JAVA der Typ bei Deklarationen ohne spitze Klammern vorangestellt wird. Es folgt dies der Tradition von C und C++. Die spitzen Klammern in Zeile 985 kennzeichnen hier lediglich wieder *metasyntaktische Variablen*.

36.1 Klassendefinitionen

Eine einfache Klassendefinition sieht in JAVA wie folgt aus:

986 **class** <Klassenname> {
987 <Typ 1> <feld1>;
988 ...
989 <Rückgabetyp 1> <methode1>(<Parametertyp 1> <parameter 1>, ...) {...}
990 ...
991 }

Die geschweiften Klammern sind in JAVA (wie auch in C und allen syntaktisch davon abgeleiteten Sprachen) Begrenzer für Definitionen und Blöcke; sie entsprechen den Schlüsselwörtern **begin** und **end** von PASCAL. Die spitzen Klammern kennzeichnen auch hier wieder metasyntaktische Variablen (also Platzhalter für richtige Namen).



Die obige Klassendefinition teilt sich in die Angabe von *Instanzvariablen*, die in JAVA **Felder** genannt werden, und *Instanzmethoden*, die in JAVA

Instanzvariablen und Methoden

Beide werden, der Tradition von C++ folgend, zusammenfassend auch als **Member** bezeichnet, wobei sich Member (das englische Wort für ein Element einer Menge) auf die Klassendefinition (oder *Intension*; vgl. Abschnitt 7.2 in Kurseinheit 1) bezieht. Konstruktoren zählen nicht zu den Membern; sie werden in Abschnitt 36.3 behandelt.

Neben Feldern und Methoden kann eine Klassendefinition in JAVA auch geschachtelte sowie sog. **innere Klassendefinitionen** besitzen. Diese Klassendefinitionen gelten dann ebenfalls als Member der umschließenden Klasse. Sie sind vor allem dann sinnvoll, wenn man ausdrücken will, dass die Existenz der inneren Klasse ohne die der äußeren sinnlos wäre. So ist es beispielsweise üblich, wenn man verzeigerten Listen implementiert, die Klasse der Listenelemente, deren Instanzen neben dem eigentlichen Inhalt auch noch einen oder mehrere Zeiger aufnehmen müssen, innerhalb der Klasse der Liste zu definieren, denn diese Listenelemente wird man kaum außerhalb einer Liste verwenden (und ihre Klasse auch gar nicht kennen) wollen. Innere Klassen werden zudem immer im Kontext von Instanzen ihrer äußeren Klasse(n) instanziert; Instanzen innerer Klassen liegen somit „innerhalb“ von Instanzen ihrer äußeren Klasse(n), auf die sie mit einem **Outer this** genannten Konstrukt zugreifen können.

innere Klassen

Member können auch mit dem Zusatz **static** deklariert werden. Dabei bedeutet **static** für so deklarierte Felder und Methoden, dass sie sich nicht auf Objekte beziehen (die ja dynamische Gebilde sind), sondern auf die Klasse selbst, in der sie definiert sind. Es handelt sich also um *Klassenvariablen* und *-methoden*. Da in JAVA im Gegensatz zu SMALLTALK Klassen selbst keine Objekte, sondern während der Programmausführung dauerhaft existierende, unveränderliche Gebilde sind, die nicht in Variablen gespeichert werden können, werden als **static** deklarierte Member auch nicht dynamisch gebunden.

Klassenvariablen und -methoden in JAVA

Achtung: Man könnte im Fall von **static** deklarierten Feldern meinen, dass diese dadurch zu Konstanten würden; das ist aber nicht der Fall. Dazu dient in JAVA das Schlüsselwort **final**, das für Variablen aussagt, dass ihnen genau einmal ein Wert zugewiesen werden darf. Im Gegensatz dazu brauchte man im schlüsselwortlosen SMALLTALK für Konstanten noch *konstante Methoden* (Abschnitt 4.3.6). Auf Methoden angewandt bedeutet **final**, dass diese nicht in Subklassen überschrieben werden dürfen; mehr dazu im nächsten Abschnitt.

„statisch“ heißt nicht „konstant“

36.2 Subklassenbeziehung

Es ist in JAVA wie in SMALLTALK vorgesehen, dass alle Klassen außer **Object** von bereits existierenden ableiten. JAVA verwendet dazu das Schlüsselwort **extends**:

```
992 class B extends A {  
993     ...  
994 }
```



Es wird dadurch das Bestehen einer *Subklassenbeziehung* zwischen B und A deklariert. Die Verwendung von **extends** legt bereits nahe, dass es sich dabei zugleich um eine *Typeverweiterung* handelt, aus der (gemäß Kapitel 23) *Zuweisungskompatibilität* folgt; tatsächlich ist das auch so.

Durch Angabe einer Extends-Klausel gibt eine Klasse an, von welcher anderen Klasse sie die nicht **static** deklarierten Member erbt. Da die erbende Klasse, auch in JAVA *Subklasse* genannt, die geerbten Methoden nur invariant überschreiben darf (und geerbte Felder in ihrer Sichtbarkeit nicht reduziert werden dürfen; s. Abschnitt 39.1), ist ihr Typ automatisch Subtyp des Typs der Klasse, von der sie erbt.

Genau wie in SMALLTALK gibt es in JAVA *abstrakte Klassen*. Anders als **SMALLTALK** spendiert JAVA jedoch ein Schlüsselwort **abstract**, mit dem man eine Klasse als abstrakt und damit als nicht instanziierbar deklarieren kann. Man schreibt dazu einfach

995 **abstract class** A { ... }

Das bedeutet, dass von einer solchen Klasse keine Instanzen mehr gebildet werden dürfen (vgl. Abschnitt 10.3). Dabei ist dieses Verbot, dessen Beachtung vom Compiler überprüft wird, vollkommen unabhängig davon, ob von der Klasse Instanzen gebildet werden könnten — selbst wenn alles, was man für das Funktionieren der Instanzen benötigt, in der Klasse vorhanden ist (einschließlich Konstruktoren), darf sie nicht instanziert werden. Sehr viel häufiger ist aber der Fall, dass der Klasse die Implementierung von einer oder mehreren (bis hin zu allen) benötigten Methoden fehlt; diese Methoden werden dann in der Klasse lediglich deklariert, und zwar ebenfalls mit dem Schlüsselwort **abstract**:

996 **abstract** <Rückgabetyp> <methode>(<Parametertyp> <parameter>, ...);

Man gibt dann hinter der Methodensignatur (also dem Namen und den Parametern) keine Implementierung (in geschweiften Klammern) an, sondern lediglich ein abschließendes Semikolon. Im Gegensatz dazu musste ja in SMALLTALK eine abstrakte Methode durch einen Aufruf von **implementedBySubclass** o. ä. gekennzeichnet (s. Abschnitt 10.3 in Kurseinheit 2) werden.

Die Aufforderung, eine abstrakte Methode in einer Subklasse zu implementieren, gibt es in JAVA auch, allerdings nicht per Laufzeitfehler auf Programmausführungsebene, sondern auf Compilerebene. Wenn man nämlich von einer abstrakten Klasse (per **extends**) ableitet, dann muss die abgeleitete Klasse entweder selbst als abstrakt deklariert sein oder man muss alle abstrakten Methoden der Klasse, von der sie ableitet, mit Implementierungen versehen. Tut man das nicht, erhält man vom Compiler eine entsprechende Aufforderung.

Komplementär zur Abstract-Deklaration gibt es in JAVA auch die Möglichkeit, zu verhindern, dass von einer Klasse abgeleitet wird: Man schreibt dazu anstelle von **abstract** einfach **final**. Dasselbe gilt für einzelne Methoden, deren Überschreiben in einer Subklasse man durch eine Final-Deklaration verhindern kann. Eine

abstrakte Klassen

Ableitung von abstrakten Klassen

nicht erweiterbare Klassen



einfache, goldene Regel der objektorientierten Programmierung ist übrigens, dass man alle Klassen entweder als abstrakt oder als final deklarieren sollte; dies setzt das Prinzip der abstrakten Generalisierungen durch und vermeidet die Probleme von nur von Vererbung getriebenen Klassenhierarchien (Kapitel 9 in Kurseinheit 2 und 69 in Kurseinheit 6).

36.3 Konstruktoren

Objekte, für die es keine literale Repräsentation gibt, müssen in JAVA (wie in SMALLTALK) explizit, als Instanzen von Klassen, erzeugt werden. Dazu gibt es in JAVA eine spezielle Kategorie von Methoden, *Konstruktoren* genannt, die, anders als in SMALLTALK, keine Klassenmethoden sind, sondern zwischen Klassen- und Instanzmethoden stehen. Dabei sind Konstruktoren wie Instanzmethoden, weil in ihrem Rumpf auf alle Felder und Methoden der neu erzeugten Instanz zugegriffen werden kann (und zwar genau so, als sei der Konstruktor eine Instanzmethode, die auf der neu erzeugten Instanz aufgerufen würde). Konstruktoren können also all die Anweisungen enthalten, für die in SMALLTALK noch eine spezielle Methode `initialize` notwendig war (vgl. Abschnitt 8.2 in Kurseinheit 2). Konstruktoren sind aber auch wie Klassenmethoden, weil sie eben nicht auf einer Instanz aufgerufen werden, sondern auf der Klasse. Allerdings sieht JAVA dafür keine spezielle Klassenmethode `new` o. ä. vor, sondern verwendet den Klassennamen selbst wie einen Methodennamen. Wenn man also, von SMALLTALK kommend, etwas der Form

```
997 class A {  
998     static A new(<formale Parameterliste>) {...} // Klassenmethode  
999     ...  
1000     A a = A.new(<tatsächliche Parameterliste>);  
1001     ...  
1002 }
```

erwartet würde, findet man in JAVA stattdessen

```
1003 class A {  
1004     A(<formale Parameterliste>) {...} // Konstruktor  
1005     ...  
1006     A a = new A(<tatsächliche Parameterliste>);  
1007     ...  
1008 }
```

Wie man sieht, erlauben Konstruktordefinitionen in JAVA anders als (andere) Methoden keine Angabe eines Rückgabetypen — da die erzeugte Instanz immer eine der Klasse ist, in der der Konstruktor definiert ist, steht der Typ fest. Die Angabe des Rückgabetyps in Zeile 998 wäre also redundant.

Wenn man keinen Konstruktor definiert, nimmt JAVA stets den (impliziten) Standardkonstruktor an, der parameterlos ist und der nichts weiter tut, als eine neue Instanz der Klasse zu liefern. Außerdem werden Konstruktoren nicht vererbt; stattdessen wird der Standardkonstruktor einer Klasse beim Erzeu-

Standardkonstruktor und Superkonstruktoren



gen einer Instanz von einer ihrer Subklassen automatisch mit aufgerufen. Um dieses Verhalten zu überschreiben, kann man aus einem Konstruktor heraus einen beliebigen Konstruktor der Superklasse mittels `super` aufrufen; darin wiederum aufgerufene Methoden werden dann dynamisch gebunden, was dazu führen kann, dass von diesen Methoden auf noch nicht initialisierte Variablen zurückgegriffen wird. Tatsächlich ist der ganze Komplex Konstruktoren und Initialisierung von Variablen in JAVA recht komplex, was, da Instanziierung und Initialisierung fundamentale und für jedes Programm unverzichtbare Vorgänge sind, nicht gerade für JAVA als Anfängerinnensprache spricht.

Konstruktionen zur Objekterzeugung mit Klassenmethoden wie oben (Zeile 998) sind in JAVA übrigens auch möglich; allerdings muss eine solche Klassenmethode dann in ihrem Rumpf einen Konstruktor wie in Zeile 1006 aufrufen. Man spricht dann von der Klassenmethode auch als einer *Factory-Methode* (vgl. Abschnitt 8.3 in Kurseinheit 2); sie kann auch Instanzen anderen Typs als des deklarierten zurückgeben.

36.4 Überschreiben, Überladen und dynamisches Binden

Nun hat die Subklasse die Möglichkeit, neue Member hinzuzufügen und bereits bestehende zu redefinieren. In JAVA ist die Möglichkeit der Redefinition auf die Möglichkeit des Überschreibens eingeschränkt, was soviel heißt wie dass eine Methode mit der gleichen Signatur (bestehend aus Methodenname und formalen Parametertypen) noch einmal definiert werden kann, und zwar mit geänderter Implementierung. Auch darf die Methode den Rückgabetyp kovariant, also nach unten, abändern (vgl. dazu die Diskussion in Kurseinheit 3, Abschnitt 26.3). Dass die überschreibende Methode über die Einhaltung der Typinvarianten, die mit ihren (geerbten) Parametern verbunden sind, hinaus nur etwas tut, das mit der überschriebenen Methode kompatibel ist, dass sie also nicht mit dem erwarteten Verhalten bricht, kann durch die Sprachdefinition JAVAs nicht erzwungen werden — hier ist die Programmiererin in der Verantwortung (vgl. dazu auch die Abschnitte 52.6 in Kurseinheit 5 und 54.1 in Kurseinheit 6).

Nun kann man in JAVA auch Methoden mit gleichem Namen, aber verschiedenen Parametertypen in derselben oder einer Subklasse haben. Diese Methoden nennt man dann **überladen**. Es ist wichtig, zu verinnerlichen, dass in JAVA Überladen und Überschreiben zwei grundverschiedene Dinge sind, obwohl man in beiden Fällen nichts weiter tut als eine Methode mit bereits vorhandenem Namen noch einmal zu definieren: Beim Überladen wird eine neue Methode eingeführt, beim Überschreiben wird eine bereits bestehende redefiniert. Diese Unterscheidung spielt beim dynamischen Binden eine entscheidende Rolle.

Um das dynamische Binden JAVAs genau zu verstehen (und damit das Verhalten eines Programms vorhersagen zu können), muss man sich den Bindealgorithmus vor Augen halten. Dieser geht wie folgt vor.

Überschreiben

Überladen

dynamisches Binden

Bereits zur Übersetzungszeit wird ein dynamischer Methodenaufruf lose an eine Methodendeklaration gebunden, und zwar an genau die, die die folgenden Bedingungen erfüllt:

1. Sie hat den gleichen Namen und die gleiche Anzahl Parameter⁵⁹ wie die aufgerufene Methode.
2. Sie ist in der Klasse, die zu dem deklarierten Typ des Ausdrucks gehört, der das Empfängerobjekt liefert (nicht selten einfach eine Variable), deklariert oder wird von einer ihrer Superklassen geerbt.
3. Die deklarierten Parametertypen des Aufrufs (die deklarierten tatsächlichen Parametertypen) sind jeweils Subtypen der deklarierten Parametertypen der Methodendeklaration (der deklarierten formalen Parametertypen).
4. Es gibt keine andere Methode, die die gleichen Voraussetzungen erfüllt, deren deklarierte formale Parametertypen aber gleich weit entfernt oder näher an den Typen des Aufrufs sind (Entfernung hier gemessen als die Anzahl der Subtypen, die dazwischen liegen).

Damit ist dann die oberste (Wurzel) einer Menge von Methoden gefunden, die möglicherweise in Subklassen überschrieben wird und an eine von denen der Methodenaufruf dann dynamisch gebunden wird. Man beachte, dass die überschreibenden Methoden dieselbe Signatur haben müssen wie die überschriebene; diese Methoden bilden eine Art Familie, von der eine zur Bindung herausgesucht wird.

Zur Laufzeit wird dann nur noch der tatsächliche Typ des Empfängerobjekts (die Klasse, von der es eine Instanz ist) bestimmt. Dieser muss, aufgrund der Regeln der Zuweisungskompatibilität, ein Subtyp des deklarierten Typs des Ausdrucks sein, der das Empfängerobjekt liefert. Mit diesem tatsächlichen (auch dynamisch genannten) Typ wird dann aus der zuvor bestimmten Menge von in Frage kommenden überschriebenen Methoden die ausgesucht, die in der Klasse definiert wurde, die der Klasse des Empfängerobjekts in der Kette der Superklassen die nächste ist.

Wenn bei der Suche nach einer Methode zur Übersetzungszeit nach obigem Algorithmus (Schritt 4) eine oder mehrere andere Methodendefinitionen gleich weit von der aufgerufenen entfernt sind, dann meldet der Compiler einen sog. *Method ambiguous error*, der bedeutet soll, dass die aufgerufene Methode durch den Aufruf nicht eindeutig bestimmt ist. Man beachte, dass der Fehler durch Methodenaufrufe, nicht durch Methodendeklarationen verursacht wird; wenn man den problematischen Aufruf entfernt, gibt es auch keinen Fehler mehr. Der einfachste Fall eines solchen Fehlers ergibt sich durch den Aufruf

Method ambiguous

1009 `System.out.println(null)`

⁵⁹ Diese Bedingung ist bei Methoden mit variabler Parameterzahl aufgeweicht.



bei dem unklar ist, ob `println(null)` an die Implementierung von `println(String)`, `println(Object)` oder von `println(char[])` gebunden werden soll.

Man beachte, dass in Sprachen, in denen das dynamische Binden neben dem Typ des Empfängers auch die Typen der tatsächlichen Parameter berücksichtigt (das sog. *Multi-dispatch*), der Unterschied zwischen Überladen und Überschreiben dahinschmilzt. In JAVA muss hingegen das dynamische Binden anhand der Parametertypen genau wie in SMALLTALK über *Double dispatch* (s. Abschnitt 12.3 in Kurseinheit 1) simuliert werden.

37 Ausdrücke

Ausdrücke sind in JAVA

Ausdrücke

- Literale,
- Variablen,
- die spezielle Variable `this`,
- Operatoranwendungen,
- Feldzugriffe der Form `a.x`, wobei `a` für den Besitzer des Feldes (ein Objekt oder eine Klasse) und `x` für das Feld (die Instanz- bzw. Klassenvariable) steht (`a` kann auch durch `super` ersetzt werden),
- Methodenaufrufe der Form `a.m(...)`, wobei `a` für den Empfänger des Aufrufs steht oder für `super`, `m` für die Methode und `...` für die tatsächlichen Parameter (die wiederum Ausdrücke sind),
- Array-Zugriffe der Form `a[i]`, wobei `a` für das Array und `i` für einen Index steht (eine ganze, positive Zahl; bei mehrdimensionalen Arrays können entsprechend weitere Indizes in eckigen Klammern angegeben werden),
- Klasseninstanziierungen der Form `new <Klassename> (<Parameter>)`
- Array-Instanziierungen der Form `new <Basistypname> [<n>]`, wobei `<Basistypname>` für den Typ der Elemente steht (also z. B. `int` bei einem Array von Integern) und `<n>` für einen Ausdruck, dessen Auswertung eine ganze, positive Zahl liefert (es können auch mehrere Dimensionen angegeben werden),
- Konditionalausdrücke der Form `<Boolescher Ausdruck> ? <Ausdruck 1> : <Ausdruck 2>`, wobei in Abhängigkeit davon, ob `<Boolescher Ausdruck>` zu wahr oder zu falsch auswertet, entweder `<Ausdruck 1>` oder `<Ausdruck 2>` ausgewertet wird und das Ergebnis des Gesamtausdrucks liefert,
- Cast-Ausdrücke der Form `(<Typname>) <Ausdruck>`, wobei `<Typname>` für das Ziel des Casts steht und `<Ausdruck>` für den Ausdruck, dessen Ergebnis den mit `<Typname>` bezeichneten Typ annehmen soll, sowie



- Lambda-Ausdrücke der Form (**<Parameterdeklarationen>**) -> **<Rumpf>**, wobei **<Parameterdeklarationen>** den formalen Parameterdeklarationen einer Methode entspricht und **<Rumpf>** entweder eine einzelne Anweisung oder ein Block von Anweisungen, entsprechend einem Methodenrumpf (inkl. der geschweiften Klammern) ist. Falls nur ein Parameter deklariert wird, können die runden Klammern auch weggelassen werden.

Wie man sieht, können in JAVA Ausdrücke rekursiv aus anderen aufgebaut werden: Methodenausdrücke beinhalten einen Ausdruck, der für den Empfänger steht sowie möglicherweise weitere, die für die Parameter des Methodenaufrufs („der Nachricht“) stehen, Array-Zugriffe und -Instanziierungen beinhalten Ausdrücke zur Bestimmung des Indexes bzw. der Größe, etc. Dabei müssen die Ausdrücke alle korrekt typisiert sein in dem Sinne, dass der Typ jedes Ergebnisses eines inneren Ausdrucks mit dem der Stelle des äußeren, an der er eingesetzt wird, zuweisungskompatibel sein muss.

Genau wie in SMALLTALK, aber anders als in einigen anderen objektorientierten Programmiersprachen (z. B. C++, C#) gibt es in JAVA nur *Call by value* und kein *Call by reference*; Methoden, die einem tatsächlichen Parameter einen anderen Wert zuweisen (wie z. B. eine Methode, die den Inhalt zweier Variablen tauscht), sind daher in JAVA nicht möglich.⁶⁰ Dies stellt eine erhebliche Einschränkung dar.

Parameterübergabe beim Methodenaufruf

Die Lambda-Ausdrücke von JAVA, die mit Version 8 eingeführt wurden, ersetzen die bis dahin geübte Praxis, Funktionen über anonyme innere Klassen, die ein Interface (s. Abschnitt 40) mit nur einer Methode implementieren, in Objekte zu verpacken. Geblieben ist die Tatsache, dass Lambda-Ausdrücke den Typ eines Interfaces haben, der allerdings nicht direkt angegeben, sondern inferiert wird, und dass die Funktionen, die die Lambda-Ausdrücke darstellen, über das Interface einen Namen erhalten. „Anonyme“ Funktionen oder Blöcke wie in SMALLTALK können in JAVA über vordefinierte Interfaces erstellt werden: So liefert

Typ von Lambda- Ausdrücken



```
1010 Function<Integer, Integer> f = x -> x * x;
```

ein Objekt vom (Interface-)Typ **Function**, der wie folgt deklariert ist (zur Bedeutung von **<T, R>** s. Kapitel 43):

```
1011 public interface Function<T,R> { R apply(T t); }
```

Demnach heißt die Funktion des Funktionsobjekts „apply“; der Ausdruck **f.apply(2)** liefert entsprechend 4. Hierbei entspricht **apply(.)** dem aus SMALLTALK bekannten **value:** (s. Abschnitt 4.4 in Kurseinheit 1).

⁶⁰ Dies gilt auch für formale Parameter, die aufgrund des in ihrer Deklaration angegebenen Typs Referenzsemantik haben: Auch sie können nicht so deklariert werden, dass ihnen anstelle der Referenz eine Referenz auf eine Referenz übergeben wird (s. dazu auch Abschnitt 4.3.2).



Ähnlich wie in SMALLTALK Blöcke können Lambda-Ausdrücke in JAVA Kontext einfangen und mitnehmen (vgl. Abschnitt 4.4.1 in Kurseinheit 1). Allerdings müssen die freien (lokalen) Variablen, die aus dem Kontext eingefangen werden können, `final` deklariert sein (oder zumindest `final` deklariert sein können, also ihren Wert nicht mehr ändern) und ein Rückspringen aus dem Home context mittels `return` aus dem Lambda-Ausdruck ist nicht vorgesehen.

Abgesehen von der Typisierung sind die wesentlichen Unterschiede zu SMALLTALKs Ausdrücken (Abschnitt 4.1 in Kurseinheit 1) die folgenden:

wesentliche Unterschiede zu SMALLTALK

- In JAVA gibt es auf Objekten direkte Feldzugriffe (`a.x`).
- In JAVA gibt es keine indizierten Instanzvariablen — diese werden durch Arrays als eigenständige Objekte ersetzt. Array-Objekte können über keine benannten Instanzvariablen verfügen.
- In JAVA wird zwischen Operatoranwendungen (`+`, `-` etc.), Methodenaufrufen, dem Aufruf von Konstruktoren und Arrayzugriffen unterschieden — in SMALLTALK gibt es nur Methodenaufrufe.

38 Anweisungen, Blöcke und Kontrollstrukturen

Genau wie in SMALLTALK werden in JAVA Ausdrücke im Rahmen der Ausführung von Anweisungen ausgewertet. Anders als in SMALLTALK gibt es in JAVA jedoch eine Vielzahl von Schlüsselwörtern, die Anweisungen einleiten. Dennoch ist es auch in JAVA möglich, bestimmte Ausdrücke zu Anweisungen zu machen: Man schließt einfach einen betreffenden Ausdruck durch ein Semikolon ab. Insbesondere werden so Variablendeklarationen, Wertzuweisungen, Methodenaufrufe und Klasseninstanziierungen (s. Kapitel 37) direkt zu Anweisungen. Man beachte, dass, anders als das Semikolon in PASCAL oder der Punkt in SMALLTALK, das Semikolon in JAVA kein Trennzeichen, sondern Teil der Anweisung ist.

Nahezu alle Anweisungen finden sich in JAVA-Programmen innerhalb von Methoden.⁶¹ Blöcke sind in JAVA lediglich (in geschweifte Klammern eingefasste) Abschnitte des Quelltextes, die an die Stelle einzelner Anweisungen treten können und die einen Sichtbarkeitsbereich für darin enthaltene Variablendeklarationen darstellen. Blöcke wie in SMALLTALK kennt JAVA erst seit Version 8, als *Lambda-Ausdrücke*.

Es sind also Variablendeklarationen, Methodenaufrufe (inkl. der Konstruktoraufrufe) und Zuweisungen Anweisungen. Alle anderen Anwei-

Schlüsselwörter für Kontrollstrukturen

⁶¹ Ausnahmen sind Felddeklarationen sowie Anweisungen in Initializern und Finalizern.



sungen werden durch Schlüsselwörter eingeleitet und realisieren allesamt Kontrollstrukturen, die den Kontrollfluss eines Programms dazu bringen, von der normalen, sequentiellen Ausführung abzuweichen. Im einzelnen sind dies

- die If-Anweisung der Form `if (<Boolescher Ausdruck>) <Statement>`, bei der `<Statement>` genau dann ausgeführt wird, wenn `<Boolescher Ausdruck>` zu `true` auswertet;
- die If-else-Anweisung der Form `if (<Boolescher Ausdruck>) <Statement 1> else <Statement 2>`, bei der `<Statement 1>` genau dann ausgeführt wird, wenn `<Boolescher Ausdruck>` zu `true` auswertet, und `<Statement 2>` sonst;
- die Switch-Anweisung der Form

```
switch (<Ausdruck>) {  
    case <Literal 1> : <Anweisungsliste 1>  
    case <Literal 2> : <Anweisungsliste 2>  
    ...  
    default: <Anweisungsliste>  
}
```

wobei `<Ausdruck>` sowie `<Literal 1>`, `<Literal 2>` etc. alle vom Typ `char`, `byte`, `short`, `int` (bzw. einem der dazugehörigen Wrapper-Typen), `String` oder von einem Aufzählungstyp sein müssen und `<Anweisungsliste 1>` etc. für Folgen von Anweisungen stehen können, die jeweils mit einem `break ;` abgeschlossen werden können (aber nicht müssen);

- die While-Anweisung der Form `while (<Boolescher Ausdruck>) <Statement>`, die im wesentlichen der If-Anweisung entspricht mit dem Unterschied, dass `<Statement>` nicht nur einmal ausgeführt wird, sondern solange, bis `<Ausdruck>` zu `false` auswertet;
- die Do-Anweisung der Form `do <Statement> while (<Boolescher Ausdruck>)`, die im wesentlichen dem While-Statement entspricht mit dem Unterschied, dass `<Ausdruck>` immer erst nach Ausführung von `<Statement>` ausgewertet wird (man beachte, dass Statement kein Block sein muss; das das Statement abschließende Semikolon wirkt dann etwas deplaziert (so wie das vor `else` beim If-Statement));
- die For-Anweisung in der Form

```
for (<Initialisierungsausdruck>; <Boolescher Ausdruck>;  
<Veränderungsausdruck>) <Statement>62
```

⁶² Man beachte, dass das Semikolon hier ein Trennzeichen ist und keine Anweisung abschließt. Gleichwohl können in einem der drei Segmente innerhalb der Klammern auch mehrere Ausdrücke erscheinen, die dann aber durch Kommata getrennt werden. All diese syntaktischen Inkonsistenzen sind Erbe von C.



die <Statement> solange ausführt, bis der Boolesche Ausdruck zu `true` auswertet (auf die schier unendlichen Möglichkeiten, was sich alles in <Initialisierungsausdruck> und <Veränderungsausdruck> unterbringen lässt, gehen wir hier nicht ein; traditionell wird im Initialisierungsausdruck jedoch ein Anfangswert für eine Laufvariable⁶³ gesetzt, der dann im Veränderungsausdruck modifiziert, nicht selten hochgezählt, wird);

- die (erweiterte) For-Anweisung in der Form `for (<Variable> : <Ausdruck>) <Statement>`, die <Statement> für alle Werte, die <Ausdruck> liefert, einmal ausführt, und zwar mit dem jeweiligen Wert als Inhalt der Variable (wobei <Ausdruck> zu diesem Zweck entweder vom Typ eines Arrays sein oder das Interface `Iterable` implementieren muss, was soviel bedeutet wie dass das Objekt, zu dem <Ausdruck> auswertet, die Methoden `hasNext()` und `next()` anbieten muss);
- die Break-Anweisung der Form `break;` bzw. `break <Label>;`, die innerhalb von Schleifen oder Switch-Statements dazu führt, dass diese sofort verlassen werden, wobei <Label> sich auf ein Label bezieht, das einer äußeren Schleife oder einem äußeren Switch-Statement vorangestellt wurde;
- die Continue-Anweisung der Form `continue;` bzw. `continue <Label>;`, die innerhalb von Schleifen dazu führt, dass der Rest des innersten bzw. des durch <Label> bezeichneten Schleifenrumpfs für den aktuellen Durchlauf nicht mehr ausgeführt wird, sondern sofort mit dem nächsten Durchlauf, falls vorhanden, weitergemacht wird (Continue-Anweisungen außerhalb von Schleifen bzw. mit einer Nicht-Schleife als Label sind ein Syntaxfehler);
- die Return-Anweisung der Form `return;` bzw. `return <Ausdruck>;`, die bewirkt, dass die umschließende Methode sofort beendet und ggf. der Wert der Auswertung von <Ausdruck> zurückgegeben wird (`return;` darf auch in einem Konstruktor vorkommen);
- die Synchronized-Anweisung der Form `synchronized (<Ausdruck>) <Block>`, die dafür sorgt, dass der durch <Block> bezeichnete Anweisungsblock nur ausgeführt wird, wenn das mit dem Objekt, zu dem <Ausdruck> auswertet, verbundene Lock dies zulässt (s. Abschnitt 47.3);
- die Try-Anweisung der Form

```
try <Try-Block>
  catch (<formaler Parameter 1>) <Catch-Block 1>
  catch (<formaler Parameter 2>) <Catch-Block 2>
  ...
finally <Finally-Block>
```

⁶³ Wird die Laufvariable erst im For-Statement deklariert, dann ist ihre Sichtbarkeit auf das For-Statement, einschließlich des enthaltenen Statements <Statement>, beschränkt.



wobei <Try-Block> für einen Block steht, von dessen Ausführung man weiß, dass sie durch einen Laufzeitfehler abgebrochen werden kann, wobei mit den Sequenzen **catch** (<formalerParameter1>) <Catch-Block1> usw. für verschiedene Arten von Laufzeitfehlern verschiedene Behandlungsblöcke angegeben werden können und wobei **finally** <Finally-Block> einen Block zu spezifizieren erlaubt, der immer ausgeführt wird, nachdem alle anderen Blöcke ausgeführt (oder abgebrochen) wurden (kann auch weggelassen werden);

- die Throw-Anweisung der Form **throw** <Exception>; die das Programm eine Exception werfen lässt;
- die Assert-Anweisung der Form **assert** <Ausdruck 1>; oder **assert** <Ausdruck 1> : <Ausdruck 2>;, wobei <Ausdruck 1> ein Boolescher Ausdruck und <Ausdruck 2> von beliebigem Typ außer dem von **void** sein muss, mit der Bedeutung, dass wenn <Ausdruck 1> zu **false** auswertet, dass dann das Programm mit einer entsprechenden Fehlermeldung abgebrochen wird, wobei ggf. das Ergebnis von <Ausdruck 2> mit der Fehlermeldung ausgegeben wird.

Außerdem ist die leere Anweisung, bestehend aus einem einzelnen Semikolon, eine Anweisung.

Bemerkungen:

- Das Weglassen von **break**; am Ende einer Liste von Anweisungen in einem Case-Zweig der Switch-Anweisung wird als *Fall through* bezeichnet und bewirkt, dass mit den Anweisungen des nächsten Case-Zweigs fortgefahrene wird. Dies ermöglicht, mehrere Fälle zusammenzuführen, ist aber eines der fehlerträchtigsten Konstrukte C-artiger Sprachen.
- Die Assert-Anweisung kann *Seiteneffekte* haben, also z. B. die Werte von Variablen aus umgebenden Blöcken oder von Instanzvariablen ändern. Wenn der Ablauf des Programms von diesen Werten abhängt, macht es einen Unterschied, ob ein Programm mit oder ohne Prüfung der Assertions ausgeführt wird. Das ist ein starkes Stück.



- Assert-Anweisungen sind ein erster zarter Versuch, in JAVA auch noch andere Invarianten als die Typinvarianten unterzubringen. Dabei findet die Überprüfung dieser, mittels `assert` eingebrachten Invarianten im Gegensatz zum Großteil der Typprüfung erst zur Laufzeit statt, indem nämlich die entsprechenden Statements ausgeführt werden. Dabei beziehen sich die Bedingungen, die die Invarianten formulieren, häufig auf Methoden des Programms, für die die Invarianten angegeben werden sollen. Da diese Methoden aber auch den Zustand des Programms ändern können, kann man einer Zusicherung mit `assert` nicht ansehen, ob sie seiteneffektfrei ist. Fortschrittlichere Verfahren zur Zusicherung von Invarianten werden nicht als Anweisungen formuliert, sondern als Quellcodeannotationen, und stellen zudem sicher, dass alle Zugriffe auf Programmelemente, die zur Laufzeit notwendig sind, den Zustand des Programms nicht verändern (mehr zu diesem Thema im Kurs 01853 sowie in Abschnitt 52.6 von Kurseinheit 5).



39 Module

Ein **Modul** ist eine Einheit von Programmelementen, die (bzw. deren Funktion) von außen (also z. B. von anderen Modulen) nur über die *Schnittstelle des Moduls* zugänglich sind. Damit behält ein Modul einen Teil seiner Implementation für sich — es hütet quasi ein *Implementationsgeheimnis*. Der Teil, den es (über seine Schnittstelle) nach außen trägt, gilt gemeinhin als öffentlich. Die „Öffentlichkeit“ kann dabei durchaus beschränkt sein (s. dazu auch Abschnitt 52.2 in Kurseinheit 5).

Ein wesentlicher Grund, zwischen öffentlichen und nicht öffentlichen — privaten — Teilen eines Moduls zu unterscheiden, ist, dass die Programmiererinnen eines Moduls Hoheit darüber behalten wollen, wie sie das Modul programmieren. Indem sie sich auf eine Schnittstelle festlegen und alles andere hinter der Schnittstelle verbergen, ist es ihnen möglich, jederzeit die verborgenen Teile zu ändern, ohne dass irgendeine andere davon in Kenntnis gesetzt werden muss — weil die privaten Teile von außen unsichtbar sind, hängt auch niemand davon ab, und insofern ist auch niemand davon betroffen, wenn an einem Modul eine Änderung durchgeführt wird, die die Schnittstellen unberührt lässt.

öffentliche und private Teile eines Moduls

In JAVA wurden bis Version 8 Module durch Klassen und Pakete (engl. packages) mehr oder weniger gut simuliert. Mit JAVA 9 wurde schließlich — nach langer Vorbereitungszeit — ein Modulbegriff eingeführt, der diesen auch Namen verdient.

Link



39.1 Klassen und Pakete als Module

Klassen haben in JAVA mehrere Funktionen: Neben der offensichtlichen, der Vorlage für die Erzeugung von Objekten, liefern sie auch noch Typen eines Programms und dienen der Modularisierung.



Pakete hingegen dienen der Sammlung von Klassen und sind zugleich Namensräume für sie (keine zwei Klassen innerhalb eines Pakets dürfen gleich heißen). Außerdem gelten für Klassen innerhalb einer Pakets laxere gegenseitige Zugriffsbeschränkungen als für Klassen aus verschiedenen Paketen. Zwar können Pakete hierarchisch organisiert sein, aber diese Organisation hat keine Bedeutung. Insbesondere erlauben Pakete keinen privilegierten Zugriff auf Klassen ihrer Subpakete. Damit Klassen paketübergreifend aufeinander zugreifen können, bedarf es expliziter Import-Deklarationen unter Nennung der jeweiligen Paketnamen.

Die Zugreifbarkeit von Klassen oder Typen und den Elementen ihrer Definition (den Membern) wird in JAVA durch sog. *Zugriffsmodifikatoren*

Zugriffsmodifikatoren

(engl. access modifiers) eingeschränkt.⁶⁴ Es sind dies **private**, **protected**, **public** sowie das sog. Package local, für das es kein Schlüsselwort gibt und das in Klassen bei Fehlen eines der drei anderen angenommen wird (bei Interfaces wird **public** angenommen). Innerhalb der Klasse selbst sind alle Elemente seiner Definition zugreifbar, innerhalb ihrer Subklassen die, die **public** oder **protected** deklariert wurden, innerhalb der Klassen desselben Pakets alle, die **public**, **protected** oder ohne Zugriffsmodifikator deklariert wurden, und in anderen Paketen nur noch die, die **public** deklariert wurden. Faktisch werden Typdefinitionen somit relativ: Was ein Typ anbietet, hängt nicht nur vom Typ selbst ab, sondern auch davon, wo er verwendet wird. Konzeptionell hat die mit einem Typ gemeinsam deklarierte Zugriffsbeschränkung jedoch nichts mit dem Typ zu tun, sondern ist vielmehr die *Schnittstellenspezifikation eines Moduls*, wobei das Modul die Klasse ist.

Auf die Bedeutung von Klassen als Module gehen ich in Kapitel 59 von Kurseinheit 6 noch genauer ein. Hier schauen wir uns noch kurz die konkreten Auswirkungen der Zugreifbarkeitsbeschränkungen an:

Beispiel für Zugriffsmodifikatoren und deren Auswirkungen

```
1012 package a;
1013 public class A {
1014     public int i;
1015     protected int j;
1016     int k;
1017     private int l;
1018     public int m() {...}
1019     protected int n() {...}
1020     int o() {...}
1021     private int p() {...}
1022 }

1023 package a;
1024 public class B {}

1025 package a.a;
1026 public class C {}

1027 package a.a;
```

⁶⁴ Die Zugreifbarkeit in JAVA wird oft, und dabei irrtümlich, auch als *Sichtbarkeit* (visibility) bezeichnet. Sichtbarkeit bezieht sich in JAVA aber auf lexikalische Scopes; sie wird durch Hiding, Shadowing und Obscuring eingeschränkt (und kann dann bisweilen über Qualifizierer wiederhergestellt werden).



Auf einer per A a deklarierten Variable sind abhängig davon, innerhalb welcher Klasse die Deklaration steht, die folgenden Elemente zugreifbar (ergeben die folgenden Ausdrücke keine Typfehler):

- in Klasse A selbst: alle
- in Klasse B: a.i, a.j, a.k, a.m(), a.n() und a.o()
- in Klasse C: a.i und a.m()
- in Klasse D: a.i, a.j, a.m() und a.n()

Die Verquickung von Typ und Zugriffsbeschränkung (Schnittstelle) ist etabliert und kommt auch in anderen Sprachen vor (z. B. EIFFEL, C# und C++). Sie hat den Vorteil der sprachlichen Knappheit (Typ- und Schnittstellendeklaration in einem) und den Nachteil, dass die Zugriffsbeschränkungen nur sehr grob eingestellt werden können — insbesondere ist es nicht möglich, dass sich zwei (inhaltlich eng zusammengehörende) Klassen gegenseitig einen freieren Zugriff gestatten als allen anderen, es sei denn, man packt diese beiden in ein Paket.⁶⁵ Außerdem hat sie den Nachteil, dass zwei unterschiedliche Konzepte der Programmierung zusammen geworfen und dadurch von Programmiererinnen u. U. nicht mehr als unterschiedlich wahrgenommen werden. JAVA-Compiler unterscheiden aber immerhin zwischen Typfehlern und Zugriffsfehlern (z. B. „is undefined“ vs. „is not visible“ in ECLIPSE, wobei letzteres freilich „is not accessible“ heißen müssen).

Vor- und Nachteile der Verquickung von Typ und Zugriffsbeschränkung

39.2 Abhängigkeiten zwischen Modulen

Die Aufteilung eines Programms auf Module dient vor allem dem Zweck der unabhängigen Entwicklung der Programmteile. Damit dies erreicht wird, muss die **Abhängigkeit** der Module möglichst gering ausfallen. Dabei bedeutet Abhängigkeit in der Regel **Änderungsabhängigkeit**: Wenn sich in einem Teil etwas ändert, muss sich auch im davon abhängigen Teil etwas ändern. Sie ergibt sich regelmäßig aus einer Benutzt-Beziehung; in der objektorientierten Programmierung kommt jedoch noch die **Vererbungsabhängigkeit** hinzu. Module, die vollkommen unabhängig voneinander sind, sind ein Indikator dafür, dass man nicht ein Programm entwickelt, sondern mehrere — wo keine Abhängigkeiten bestehen, da gibt es auch kein Zusammenspiel.

Abhängigkeit ist eine gerichtete Beziehung: Dass A von B abhängt heißt nicht, dass auch B von A abhängt. Und so manifestieren sich auch in den

Manifestation der Abhängigkeit zwischen Modulen

⁶⁵ Benötigt wird der freiere Zugriff auch bei der Realisierung sog. *Multi-Methoden* (Methoden, deren Aufruf auf mehr als nur dem Empfängerobjekt *gedispatcht* wird; vgl. dazu auch Abschnitt 12.3 in Kurseinheit 1).



Klassen als Modulen der JAVA-Programmierung die zwei Richtungen von Abhangigkeit in zwei verschiedenen Formen:

1. Dass eine Klasse von (der Schnittstelle) einer anderen abhangt, erkennt man an der Tatsache, dass auf Objekte der anderen Klasse zugegriffen wird, was man wiederum daran erkennt, dass Variablen des dazugehorigen Typs deklariert werden sowie, soweit sich die andere Klasse in einem anderen Paket befindet, die Klasse oder gleich das ganze Paket importiert wird.
2. Dass eine Klasse manche Teile ihrer Member anderen zur Benutzung anbietet und diese damit von ihr abhangig werden konnen, erkennt man an der Verwendung von anderen Zugriffsmodifikatoren als **private** sowie an der Implementierung von Interfaces.

In JAVA sind zirkulare Abhangigkeiten zunachst erlaubt. Insbesondere durfen sich zwei Klassen (genauer: *Compilation units*) wechselseitig importieren. In der Softwareentwicklung ist dies jedoch verpont, schon weil durch eine wechselseitige Abhangigkeit eine enge Kopplung dokumentiert wird, die zwischen Modulen grundsatzlich nicht bestehen sollte. Fur JAVA-Module (s. u.) sind zirkulare Abhangigkeiten denn auch verboten. Wenn Sie einmal in die Verlegenheit kommen sollten, selbst JAVA-Module zu definieren, werden Sie feststellen, dass dieses Verbot eine sehr sorgfaltige Planung der Modularisierung erzwingt (was fur sich genommen schon einen Wert darstellt).

zirkulare Abhangigkeiten



WIKIPEDIA

Vererbungsabhangigkeiten zwischen Klassen, die in JAVA durch Verwendung des Zugriffsmodifikators **protected** und der Annotation **@Override** zumindest angedeutet werden (mehr dazu in Kurseinheit 6, Kapitel 55), sind naturgem nicht zirkular; bei der Aufweitung der Abhangigkeit von Klassen auf Pakete konnen jedoch auch zirkulare Abhangigkeiten entstehen.

39.3 Die Module von JAVA 9

Als JAVA entworfen wurde, ging man wohl davon aus, dass ein Programm aus mehreren Klassen besteht, die alle zu einem Paket zusammengefasst werden konnen. Die Klassen eines Programms gewahren sich somit untereinander privilegierten Zugriff (alles, was nicht **private** deklariert ist, ist zugreifbar), nach auen sind jedoch nur **public** deklarierte Programmelemente sichtbar.

Dieser Ansatz funktioniert jedoch in dem Moment nicht mehr, in dem Programme auf mehrere Pakete aufgeteilt werden. Wenn zwischen den Paketen namlich Abhangigkeiten bestehen (was, wenn die Pakete zusammen *ein* Programm representieren, naturgem der Fall



ist), dann sind hierfür `public` Deklarationen erforderlich, die die so deklarierten Programmlemente jedoch für alle gleichermaßen sichtbar machen.⁶⁶ Ein Programm kann also keine andere (eingeschränktere) öffentliche Schnittstelle haben als die Summe seiner Pakete.

Diese unbefriedigende Situation wurde von der JAVA-Community aufgegriffen, die mit der *Open Services Gateway Initiative (OSGi)* einen Standard etablierte, der die Zusammenfassung von JAVA-Paketen zu Modulen mit einer eigenen Schnittstellenspezifikation erlaubte. Dieser Standard ist u. a. Grundlage von Eclipse, das nicht nur eine JAVA-IDE, sondern im Kern ein Framework für die Entwicklung beliebig komplexer JAVA-Programme (sog. Rich Clients) ist.



Mit JAVA 9 wurde dann JAVA ein eigenes, über Klassen und Paketen stehendes Modulkonzept verpasst. Ein Modul ist demnach eine Menge von Paketen, die, über eine Moduldeklaration, eine gemeinsame Schnittstelle spezifizieren. Eine Moduldeklaration besteht aus einem (eindeutigen) Namen des Moduls, einer Deklaration der angebotenen Schnittstelle (bislang über `public` Deklarationen hergestellt) und einer Deklaration der benötigten Schnittstelle (bislang ausschließlich über `import` Direktiven deklariert). Neben den allgemeinen Export tritt der sog. qualifizierte Export, wie er auch in EIFFEL vorgesehen ist: Er nennt die Module, an die exportiert wird, namentlich und erlaubt so einen „privaten“ Austausch zwischen bestimmten Modulen (die somit ihre eigenen, oder privaten, Schnittstellen haben).

Das besondere an der Moduldefinition von JAVA ist, dass die Einhaltung der damit einhergehenden Schnittstellenspezifikationen sowohl zur Übersetzungszeit als auch zur Laufzeit erzwungen wird. Außerdem kontrolliert sie auch den *reflektiven Zugriff* auf Programmelemente, der bislang gar nicht unterbunden werden konnte. JAVA-Programme werden dadurch erheblich sicherer.

40 Interfaces

Modul und Interface sind eigentlich ein Begriffspaar — das eine lässt sich nur mithilfe des anderen definieren. Die Interfaces in JAVA sind jedoch durchaus eigenständige Konstrukte, die inzwischen weit über die eigentliche Bedeutung des Begriffs, nämlich eine Schnittstelle zu definieren, hinausgehen. Auch wenn JAVAs Interfaces Vorgänger haben, betrachte ich sie doch als einen von JAVAs wichtigsten Beiträgen für die Entwicklung objektorientierter Programmiersprachen.



⁶⁶ Dies ist beispielsweise in C++ anders: Dessen Friend-Konstrukt erlaubt den privilegierten Zugriff zwischen Klassen jenseits derer öffentlichen und privaten Schnittstellen (s. Abschnitt 51.3 in Kurseinheit 5).



40.1 Interfaces als Schnittstellen

Die öffentliche Schnittstelle einer Klasse in JAVA ist die Menge ihrer Instanzvariablen und -methoden (in JAVA zusammen auch *Member* genannt), die `public` deklariert sind. Nun gibt es in JAVA die Möglichkeit, eine öffentliche Schnittstelle als eigenständiges Konstrukt zu deklarieren, das von dem der Klasse unabhängig ist, das aber genauso wie eine Klasse einen Typ definiert. Es geschieht dies mit Hilfe des Schlüsselwortes `interface`:

```
1029 interface <Interfacename> {  
1030   <Rückgabetyp 1> <methode>(<Parametertyp 1> <parameter 1>, ...);  
1031   ...  
1032 }
```

Anders als bei Klassen entspricht hier das Fehlen eines Zugriffsmodifikators der Zugreifbarkeit `public` — alles andere scheint für eine Schnittstelle auch unsinnig. Folgende weitere syntaktischen Unterschiede der Interfacedeklaration in den Zeilen 1029–1032 zur Klassendefinition der Zeilen 986–991 fallen auf:

- die Verwendung des Schlüsselwortes `interface` anstelle von `class` (klar),
- das Fehlen von Felddeklarationen und
- der Umstand, dass der Methodendeklaration keine Implementierung mehr folgt, sondern lediglich das abschließende Semikolon.

Das Schlüsselwort `abstract`, das eine Klasse für eine solche Methodendeklaration anführen müsste, wird in Interfaces automatisch angenommen; Interfaces liefern zunächst erwartungsgemäß weder Implementierungen noch Objekte (Instanzen). Mit derselben Begründung kann auch das Fehlen von Feldern erklärt werden: Da bei Feldern nicht zwischen Deklaration und Implementierung (*Definition*; s. Kapitel 19 in Kurseinheit 3) unterschieden werden kann (die Deklaration ist, da sie Namen und Typ vorgibt und mehr auch für eine Implementierung nicht angegeben werden kann, zugleich Implementierung des Feldes), wurden sie aus den Interfaces verbannt. Diese Einschränkung ist aber keine wirkliche, da ein Feldzugriff in einem Interface durch *Zugriffsmethoden* (Accessoren, also durch Getter und Setter) ersetzt werden kann.

Eine Klasse kann nun angeben, dass sie ein Interface implementiert. Sie tut das unter Verwendung des Schlüsselwortes `implements`:

Interface-
implementation und
Typkonformität

```
1033 class <Klassenname> implements <Interfacename> {...}
```

Damit verpflichtet sich die Klasse, alle im Interface angekündigten Methoden zu implementieren und öffentlich anzubieten.⁶⁷ Dabei kann eine Klasse mehrere Interfaces gleichzeitig implementieren (die entsprechenden Namen werden einfach, durch Kommata getrennt,

⁶⁷ Sie muss sie nicht unbedingt selbst implementieren, sondern kann das auch ihren Subklassen überlassen; in diesem Fall muss die Methode aber zumindest als abstrakt deklariert werden.



aneinandergehängt); zugleich ist die Angabe der implementierten Interfaces eine *nominale Typkonformitätsdeklaration*, d. h., Instanzen der Klasse sind mit allen Variablen jedes der genannten Interfaces zuweisungskompatibel.

Es definieren also die Interfaces von JAVA genau wie Klassen Typen und können daher genauso wie Klassen in Variablen-deklarationen verwendet werden:

Interfaces in Variablen-deklarationen

1034 <Interfacename> <Variablenname>

ist eine solche Deklaration. Der Type checker garantiert dann, dass auf der Variable mit Namen „Variablenname“ nur die Methoden aufgerufen werden können, die im Interface mit Namen „Interfacename“ deklariert sind, selbst wenn das Objekt, das die Variable benennt, mehr anbietet. So ist es möglich, dass Methoden und Felder einer Instanz vor anderen Instanzen anderer oder derselben Klasse verborgen werden können: Man deklariert einfach die Variablen, die auf die Instanz verweisen, mit dem Interface als Typ. Eine genauere Betrachtung der Bedeutung der Verwendung von Interfaces erfolgt in Kapitel 45.

Es kann also eine Klasse in JAVA zwar nur direkte Subklasse genau einer anderen Klasse sein, dafür aber mehrere Interfaces gleichzeitig implementieren. Diese mögliche „Mehrfachimplementierung“ von Interfaces wurde häufig als Ersatz für die in JAVA fehlende Möglichkeit der Mehrfachvererbung angepriesen — das aber war Unsinn, denn bei der Implementierung eines Interfaces wurde nichts vererbt (sieht man mal von der sog. *Interfacevererbung* ab, die aber auch keine wirkliche Vererbung ist, denn auch die Deklarationen werden nicht automatisch von einem Interface auf seine implementierenden Klassen übertragen, sondern müssen dort wiederholt werden). Vielmehr hat man es mit einer Art Mehrfach-Subtyping zu tun, das aber auch ganz nett ist, wie die Überlegungen in Kapitel 45 zeigen werden.

„Interfacevererbung“ ist keine Vererbung

40.2 Interfaces als abstrakte Klassen

Wenn ich im vorangegangenen Absatz das Präteritum bemüht habe, dann liegt das daran, dass Interfaces in JAVA seit Version 8 einen Bedeutungswandel erfahren haben. Der ursprüngliche Anlass hierfür ergibt sich aus der Evolution von Software, genauer aus der Erweiterung von Interfaces im Laufe der Zeit um zusätzliche Methoden und der Tatsache, dass solche Erweiterungen ein Nachführen der Klassen, die die Interfaces implementieren, zwingend erfordern. Insbesondere bei sog. Black-box-Frameworks, deren Interfaces dazu gedacht sind, von Anwendungsklassen implementiert zu werden, deren Entwicklung in den Händen Dritter liegt, ist dies ein erhebliches Problem. Dasselbe Problem hätte man nicht, wenn man statt der Interfaces abstrakte Klassen nehmen würde — dann könnte man nämlich die zusätzlichen Methoden mit einer Default-Implementierung versehen, die von den „implementierenden“ Klassen geerbt würde, sofern diese Klassen keine eigenen Implementierungen angeben.



Genau das wurde in JAVA 8 auch für Interfaces eingeführt: Ein Interface

Default-Methoden

kann die Implementierung einer Methode vorgeben, die dann an implementierende Klassen (oder per `extends` abgeleitete Interfaces) vererbt wird. Dafür hat JAVA ein neues Schlüsselwort spendiert bekommen: `default`. Allerdings haben Interfaces immer noch keine Instanzvariablen und entsprechend können **Default-Methoden** allenfalls auf abstrakte `Getter` und `Setter` zugreifen.

Mit JAVA 9 schließlich wurde — letztlich nur konsequent — erlaubt, in

private Methoden

Interfaces auch private Methoden zu definieren. Da diese ausschließlich aus dem Interface selbst heraus zugreifbar sind, dienen sie ausschließlich der Verbesserung der Lesbarkeit von Default-Methoden, indem man zusammenhängende Teile aus ihnen herauslässt und in private Methoden verlegt.

41 Arrays

In JAVA ist es möglich, sowohl von primitiven (Wert-)Typen als auch von Referenztypen Arrays zu bilden. Anders als z. B. in PASCAL können aber über den Array-Typkonstruktor keine neuen Typen benannt werden; die Typkonstruktion erfolgt immer implizit in einer Variablendeklaration.

```
1035 float[] f;  
1036 Object[] o;
```

vereinbart in JAVA zwei Variablen, wovon die erste ein Array von Fließkommazahlen zum Typ hat und die zweite ein Array von Objekten. Die alternative Schreibweise

```
1037 float f[];  
1038 Object o[];
```

ist auch gebräuchlich. Anders als z. B. in PASCAL wird die Größe des Arrays in der Deklaration nicht festgelegt — dies geschieht erst bei der Initialisierung.

In JAVA ist es möglich, Arrays bei ihrer Deklaration zu initialisieren:

Array-Initialisierung

```
1039 float f[] = {1.0, 3.142};
```

Man beachte die Ähnlichkeit zu literalen Arrays SMALLTALKs (Abschnitt 1.2); allerdings müssen die Elemente der Arrays in JAVA nicht selbst Literale sein, sondern dürfen auch andere Ausdrücke sein. Die Größe des Arrays (in diesem Fall 2) wird bei der Initialisierung automatisch mit festgelegt; ansonsten muss dies bei der Erzeugung des Arrays mittels eines Konstruktors explizit geschehen:

```
1040 Object[] o = new Object[2];
```



Alle Elemente des Arrays enthalten danach jedoch `null` (JAVAs äquivalent von SMALLTALKs `nil`). JAVA-Arrays sind übrigens 0-basiert, was soviel heißt wie dass das erste Element den Index 0 hat. (Zur Erinnerung: In SMALLTALK hat es den Index 1.)

Array-Initialisierer können auch geschachtelt werden und somit mehrere Dimensionen umfassen:

```
1041 int integers[][][] = { {1, 2}, null, {238} }
```

beispielsweise liefert einen möglichen Anfangswert für ein zweidimensionales Array mit der Deklaration `int integers[][]` (also ein Array mit Elementtyp `int` und mit zwei Dimensionen). Wie man sieht, müssen die Größen der zweiten und aller weiteren Dimensionen nicht für jedes Element der ersten Dimension gleich viele Elemente enthalten (sog. Ragged oder Jagged arrays sind möglich; tatsächlich handelt es sich bei mehrdimensionalen Arrays in JAVA auch gar nicht um mehrdimensionale Arrays, sondern um Arrays von Arrays).

Interessanterweise haben in JAVA Array-Variablen immer und unabhängig vom Basistyp *Referenzsemantik*. Bei der Zuweisung an die Variable `f` oben wird also nicht ein ganzes Array als Kopie übergeben, sondern lediglich ein Pointer darauf. Dies hat vermutlich den Hintergrund, dass Array-Kopieroperationen sehr teuer sind und zudem selten benötigt werden. Warum auch immer, im Ergebnis kann `f` an eine Variable vom Typ `Object` zugewiesen werden. Eine Zuweisung von `f` an eine Variable vom Typ `Object[]` ist hingegen nicht zulässig — `float[]` ist kein Subtyp von `Object[]` und somit auch nicht damit zuweisungskompatibel. Man beachte übrigens, dass Arrays, selbst wenn sie wie Klassen und Interfaces Typen bilden, die automatisch Subtypen von `Object` sind, außer ihrem Basistyp (also beispielsweise `float` oder `Object`) keine weiteren Definitions-elemente anzugeben erlauben; insbesondere kann man für einen Array-Typen keine weiteren Eigenschaften (Felder oder Methoden) definieren. Allerdings ist für jedes Array die (Pseudo-)Variable `length` definiert, deren Inhalt die Größe des Arrays (Anzahl Elemente) angibt. Außerdem wird die Methode `clone()` aus `Object` so überschrieben, dass sie ein Objekt gleichen Typs, also ebenfalls ein Array des Basistyps, zurückgibt.

Referenzsemantik von Array-Variablen

Die bemerkte mangelnde Zuweisungskompatibilität von `Object[]` und `float[]` wirft natürlich sofort die Frage auf, ob denn auch die Zuweisung einer Variable vom Typ `A[]`, wobei `A` ein Referenztyp sei und damit automatisch ein Subtyp von `Object`, an eine Variable vom Typ `Object[]` unzulässig ist. Wir hatten ja in Abschnitt 29.3 von Kurseinheit 3 am Beispiel zweier Instanzen eines parametrischen Typs bemerkt, dass dies zu einem nicht ganz offensichtlichen Problem führt, das sich analog auf Arrays übertragen lässt. Die Überraschung folgt hier auf den Fuß: Die Zuweisung ist in JAVA zulässig.

Typkonformität und Subtyping von Arrays in JAVA

Am besten lässt sich dies an einem Beispiel erläutern. Man kann tatsächlich in JAVA bei Vorliegen der Deklarationen

```
1042 class Hund extends Tier {}
```

```
1043 Tier[] tiere;
```



```
1044 Hund[] hunde;
```

die Zuweisung

```
1045 tiere = hunde
```

durchführen. Die anschließende Zuweisung

```
1046 tiere[1] = new Tier()
```

führt dann in JAVA allerdings prompt zu einem *Laufzeittypfehler* (eine sog. *Array store exception*), denn **tiere** ist ja lediglich ein *Alias* auf ein Array mit Hunden, so dass die Zuweisung ein Tier anstelle eines Hundes an Arrayposition 1 setzt und das Array **hunde**, das ja per Deklaration nur Hunde zu enthalten verspricht, damit nicht mehr typkorrekt ist. Würde man die Zuweisung aus Zeile 1046 zulassen, dann würde in der Folge die scheinbar korrekte Zuweisung

Laufzeittypfehler bei Arrays

```
1047 Hund hund = hunde[1]
```

bei der **hund** ein Tier (das aus Zeile 1046) zugewiesen wird, die Typinvariante von **hund** verletzen, was ein Compiler aber beim besten Willen nicht mehr erkennen kann (und eine Programmiererin übrigens auch kaum).

Warum aber geht man dieses Risiko ein und überträgt die Zuweisungskompatibilität von Typen auf Arrays von diesen Typen? Zunächst einmal kann man festhalten, dass hier auf die Möglichkeit der *statischen Typprüfung*, die (auf Basis mangelnder Typkonformität) einen Typfehler bei der Zuweisung aus Zeile 1045 gemeldet hätte, zugunsten einer *dynamischen Typprüfung* mit möglicher Meldung eines Laufzeitfehlers verzichtet wurde. Dies tut man immer dann, wenn die statische Typprüfung Programme verhindert, die man gern schreiben möchte und die auch korrekt sein können, ohne dass dies jedoch vom Compiler garantiert werden könnte. Es ist nämlich gar nicht gesagt, dass die Zuweisung der Zeile 1045 immer zu einem Laufzeitfehler führt — nur wenn man anschließend schreibend (wie in Zeile 1046) auf das Array zugreift und dann noch mit dem falschen Typ, kommt es zu einem solchen Fehler (zu typinkorrekten Variablenbelegungen). Da man diese Bedingung aber schlecht zur Übersetzungszeit abprüfen kann, wird eben ein Laufzeittest durchgeführt. Ein klassischer Kompromiss, der diesmal zugunsten der Flexibilität beim Programmieren ausging.

Typprüfung als Kompromiss

Warum aber will man Zuweisungskompatibilität zwischen Arrays nicht gleichen Typs und damit Zuweisungen wie die in Zeile 1045 unbedingt haben? Die Antwort ist einfach: weil es Prozeduren gibt, die den (statischen) Typ der Array-Elemente nicht genau festlegen, sondern lediglich nach oben beschränken wollen. So gibt es beispielsweise in JAVA den Facettentyp **Comparable**, der wie folgt definiert ist:

Grund für das Subtyping bei Arrays in JAVA

```
1048 interface Comparable {  
1049     int compareTo(Object o);
```



Die Methode `compareTo` soll dabei einen Wert zurückgeben, der angibt, wie der Vergleich des Empfänger- mit dem Parameterobjekt ausgegangen ist. Eine Methode mit der Signatur `sort(Comparable[])` kann dann Arrays beliebiger Elementtypen zum Sortieren annehmen, solange diese nur `Comparable` implementieren und damit Auskunft über ihre relative Ordnung zu geben in der Lage sind. Da beim Sortieren die Elemente eines Arrays nicht ersetzt, sondern nur umgeordnet werden, kann dabei auch kein Typfehler von der Art der Zeile 1046 auftreten. Diese Methode `sort` ist also faktisch sicher — ein konservativeres statisches Typsystem hätte ihre Verwendung jedoch nicht zugelassen. Der eingegangene Kompromiss zwischen statischer und dynamischer Typprüfung ist also durchaus vertretbar.

42 Aufzählungstypen

Bis JAVA 1.4 waren `class`, `interface` und `[]` (Array) die einzigen Typkonstruktoren; mit JAVA 1.5 ist auch noch `enum` für Aufzählungstypen hinzugekommen, wobei diese unter die Klassentypen fallen: Neben der Angabe der Elemente der Aufzählung kann man auch noch Felder und Methoden angeben, die auf ihnen definiert sind. Damit werden die Elemente einer Aufzählung gewissermaßen zu konstanten, also zu lebenslang gültigen, Namen für Objekte einer Klasse, die den Aufzählungstyp repräsentiert.

43 Generische Typen

Analog zu der Einführung von parametrisierten Typen in Kapitel 29 von Kurseinheit 3 abstrahieren die generischen Typen JAVAs von Typen, indem sie die Referenzierung eines oder mehrerer Typen innerhalb einer Typdefinition durch Typvariablen zu ersetzen erlauben. Anders als in STRONGTALK stehen die Typvariablen in JAVA jedoch nicht in eckigen, sondern in spitzen Klammern. Da es dadurch zu Verwechslungen mit metasyntaktischen Variablen kommen kann, werden wir in diesem Kapitel keine mehr verwenden.



43.1 Einfache parametrische Typdefinitionen

So, wie auch unparametrisierte (nicht generische) Typen in JAVA nicht aus eigenständigen Typdefinitionen hervorgehen, sondern mit der Definition einer Klasse oder eines Interfaces einhergehen, so werden auch parametrische (generische) Typen nicht separat definiert, sondern sind das Produkt parametrischer Klassen- bzw. Interfacedefinitionen. Da aber die formalen Typparameter einer Klassen- bzw. Interfacedefinition durch verschiedene Typen ersetzt werden können, wird die alte 1:1-Beziehung zwischen Klassen und Typen aufgebrochen: Jede Klasse, deren Definition einen Typparameter enthält, steht tatsächlich für eine



ganze Menge von Typen, nämlich einen pro möglicher Belegung des Typparameters. Insbesondere führt die „Instanziierung“⁶⁸ einer parametrisch definierten Klasse mit einem tatsächlichen Typparameter nur zu einem neuen Typ, aber nicht zu einer neuen Klasse. Deswegen sind auch die Klassenvariablen und -methoden einer parametrischen Klasse für alle Instanzen ihrer generierten Typen gleich; Instanzvariablen und -methoden können dagegen den Typparameter als Typ verwenden und sich insofern unterscheiden.

Die klassische Anwendung generischer Typen findet man bei Collections:
Genauso, wie man in JAVA Arrays über einen bestimmten Elementtyp bil-

Anwendungsfall Collections

den kann, will man auch andere Arten von Collections über Elementtypen haben. Zu diesem Zweck verfügt JAVA ähnlich wie SMALLTALK über eine ganze Reihe von Collection-Klassen wie z. B. Sets (für Mengen) oder Maps (die JAVA-Variante von SMALLTALKs Dictionaries). Nun sind diese Collections (anders als Arrays) nicht Bestandteil der Sprachdefinition JAVAS, sondern Elemente einer Bibliothek, also in der Sprache selbst programmierte, für die allgemeine Verwendung gedachte Klassen. Da es vor JAVA 5 keine Möglichkeit gab, bei der Deklaration einer Variable mit einer Collection als Typ anzugeben, welchen Typ die Elemente der Collection haben sollen, wurde implizit davon ausgegangen, dass diese vom Typ `Object` sind. So hatte z. B. die Klasse `ArrayList` ein (privates) Feld `elementData` vom Typ `Object[]`, in dem die Elemente gespeichert wurden. Da `Object` Supertyp aller Referenztypen in JAVA ist, konnten auch Instanzen aller Referenztypen in `elementData` und somit in Instanzen von `ArrayList` gespeichert werden.

Dank der generischen Typen ist es aber möglich, bei der Deklaration einer Variable vom Typ einer Collection — in Analogie zur Deklaration einer Variable vom Typ eines Arrays über einen Elementtyp — den Elementtyp mit anzugeben. Um beispielsweise eine Liste von Integern zu deklarieren und zu initialisieren, muss man lediglich

```
1051 List<Integer> liste = new ArrayList<Integer>();
```

schreiben.⁶⁹ Die Klassendefinition von `ArrayList` ist dazu wie folgt parametrisiert:

```
1052 public class ArrayList<E> ... {  
1053     private E[] elementData;  
  
1054     public E get(int index) {  
1055         ...  
1056         return elementData[index];  
1057     }  
  
1058     public void add(E element) {
```

⁶⁸ nicht zu verwechseln mit der Instanziierung einer Klasse — insbesondere wird hier auch kein Typ aus einem Metatyp erzeugt

⁶⁹ Man beachte, dass hier als Typ der Variable ein Interface, nämlich `List`, gewählt wurde und nicht die Klasse `ArrayList`, von der ihr Inhalt eine Instanz ist. Dies hat den Vorteil, dass die Instanz auch gegen solche anderer Klassen ausgetauscht werden kann, solange diese nur ebenfalls den Typ `List` implementieren. Sie sollten sich zur Angewohnheit machen, immer den allgemeinsten verfügbaren Typ zu verwenden, solange Sie nicht eine speziellere Filterfunktion beabsichtigen.



```
1059 } ...
1060 ...
1061 ...
1062 }
```

Dabei ist das in spitzen Klammern stehende E der (formale) Typparameter der Definition von `ArrayList`. Die bereits in Abschnitt 29.1 erwähnte Konvention, einzelne Großbuchstaben für Typparameter zu wählen, wurde auch in JAVA übernommen, ganz einfach, um Typparameter von Klassen- und Variablennamen im Programmtext besser unterscheiden zu können. So steht E üblicherweise für den Elementtyp von Containern, wie es die Collections sind. Durch die Sprachdefinition erzwungen wird das jedoch nicht.

Die Zuweisung von Zeile 1051 ist übrigens nur gültig, wenn der (parametrisierte) Typ `ArrayList<Integer>` Subtyp von `List<Integer>` ist. Dies wird, in JAVA-Syntax, durch die Deklaration

**Subtyping
parametrischer
Typen**

```
1063 class ArrayList<E> implements List<E>
```

(nominales Subtyping) sichergestellt. Man beachte, dass die Variable E hier eine logische Bedingung ausdrückt, nämlich die, dass ein bei der Verwendung von `ArrayList<E>` angegebener tatsächlicher Typparameter auch in die Definition von `List<E>` eingesetzt werden muss.

Der Compiler weiß nun aufgrund der Ersetzung des formalen Parameters E mit dem tatsächlichen Parameter `Integer` in Zeile 1051, dass die Elemente der Variable `liste` alle vom Typ `Integer` sind und dass eine Zuweisung der Form

**erhöhte
Typsicherheit**

```
1064 Integer i = liste.get(1);
```

typkorrekt ist. Um das zu überprüfen, muss er nämlich nur den Wert des Typparameters in der Deklaration von `liste`, `Integer`, in die Variable E der Implementierung von `get` einsetzen. Er kann dann feststellen, dass der Rückgabetyp des Ausdrucks mit der Variable zugeisungskompatibel ist. Man beachte, dass ohne Typparameter in Zeile 1064 eine *Typumwandlung* von `Object` auf `Integer` (*Down cast*) notwendig wäre, die aber zu einem Laufzeitfehler führen kann (s. Abschnitt 44.1). Die Einführung von Generics erhöht hingegen die Typsicherheit statisch, also zur Übersetzungszeit. Ein fundamentaler Gewinn.

Nun wissen Sie ja bereits aus der Schilderung aus Kapitel 41, dass `List<Integer>` nicht unbedingt ein Subtyp von `List<Object>` sein sollte, selbst wenn `Integer` ein Subtyp von `Object` ist. Und so führt in JAVA bei generischen Typen anders als bei Arrays schon die Zuweisung

```
1065 List<Object> objektliste = liste;
```



(bei Beibehaltung obiger Deklaration von `liste`) zu einem statischen Typfehler, der schon während der Übersetzung gemeldet wird. Man hat den oben diskutierten Kompromiss offenbar nicht weiter fortführen wollen.

43.2 Parametrische Typen und Subtyping: Wildcards

Nun ist es aber für generische Typen genau wie für Arrays durchaus sinnvoll, eine liberalere Form von Zuweisungskompatibilität zuzulassen, z. B. um Objekte verschiedener Instanzen⁷⁰ eines parametrisierten Typs bei einem Methodenaufruf demselben formalen Parameter zuzuweisen. So möchte man eben auch für generische Collections eine Methode `sort` mit einem Parameter, der eine zu sortierende Liste o. ä. enthalten soll, definieren und diese dann mit Objekten verschiedener Instanzen von `ArrayList<E>` (wie in Zeile 1052 ff. definiert) aufrufen können, also z. B. mit Objekten vom `ArrayList<Integer>` und `ArrayList<String>`. Intuitiv möchte man dazu zunächst

```
1066 void sort(ArrayList<Comparable> liste) {...}
```

schreiben, aber wie wir schon gesehen haben, sind, obwohl `Integer` und `String` Subtypen von `Comparable` sind, `ArrayList<Integer>` und `ArrayList<String>` nicht zuweisungskompatibel mit `ArrayList<Comparable>`.

Was also tun? In JAVA hat man dafür das Konzept der Typ-Wildcards (zu deutsch vielleicht Typ-Joker oder -Platzhalter) eingeführt, die bei der Instanziierung eines generischen Typs den Platz des tatsächlichen Typparameters einnehmen können und dort zunächst für einen beliebigen Typ stehen. Das Symbol für das Typ-Wildcard ist das Fragezeichen: `List<?>` ist also ein Typ, mit dem Variablen (inkl. formale Parameter) deklariert werden können. Per Definition ist dieser Typ, `List<?>`, Supertyp aller Instanzierungen von `List<T>` — `List<Integer>` beispielsweise und `List<String>` sind mit `List<?>` zuweisungskompatibel. Man beachte, dass bei einer Deklaration und anschließenden Zuweisung

**Wildcards ersetzen
Subtyping**



```
1067 List<?> liste = new ArrayList<Integer>();
```

das Fragezeichen nicht durch `Integer` ersetzt wird; die anschließende Zuweisung

```
1068 liste = new ArrayList<String>();
```

ist also möglich.⁷¹

⁷⁰ s. Fußnote 68

⁷¹ Man beachte, dass die Varianz bei der Verwendung des generischen Typs bei seiner Benutzung in einer (anderen) Deklaration hergestellt wird. Man nennt dies daher auch *Use-site variance*, im Gegensatz zur *Declaration-site variance*, wie man sie beispielsweise in C# vorfindet (s. Abschnitt 50.4.3 in Kurseinheit 5).



`List<?>` ist genau ein Typ; das Fragezeichen selbst ist jedoch keiner (und ist auch keine Typvariable). Mit Typ-Wildcards parametrisierte Typen wie `List<?>` und auch `ArrayList<?>`, im folgenden **Wildcard-Typen** genannt, sind abstrakt in dem Sinne, dass es keine direkten Instanzen von ihnen gibt:

```
1069 new ArrayList<?>()
```

ist also illegal.

Nun kann man mittels Typ-Wildcards natürlich nicht die oben geschilderten Probleme im Zusammenhang mit dem Aliasing aushebeln. Es ist also insbesondere nicht möglich, nach Deklaration und Zuweisung von Zeile 1067

```
1070 liste.add(2)
```

zu schreiben, da ja nicht sichergestellt werden kann, dass Liste tatsächlich auf ein Objekt vom Typ `List<Integer>` verweist. Tatsächlich ist die einzige gültige Zuweisung an Elemente von `liste` die von `null`. Umgekehrt kann beim Lesen der Elemente aus `liste` kein anderer Typ als `Object` angenommen werden, da ja `liste` Listen mit beliebigem Elementtyp zugewiesen werden dürfen. Das aber ist unbefriedigend.

Nun kennen Sie aus Abschnitt 29.4 bereits das Konzept der Beschränkung von Typparametern. Dieses lässt sich auch auf Typ-Wildcards übertragen. Wenn man also sicherstellen will, dass die Elemente einer Liste eines unbekannten Elementtyps mindestens vom Typ `Number` sind, schreibt man in der Deklaration der entsprechenden Variable, hier wieder `liste`, einfach

nach oben
beschränkte Typ-Wildcards

```
1071 List<? extends Number> liste;
```

Es sind damit nur noch Zuweisungen von Listenobjekten an `liste` gestattet, deren Elementtyp den Typ `Number` oder einen Subtyp davon (z. B. `Integer`) hat:

```
1072 liste = new ArrayList<Integer>();
```

ist also legal,

```
1073 liste = new ArrayList<Object>();
```

hingegen nicht. Das erlaubt einer lesende Zugriffe der Form

```
1074 Number n = liste.get(1);
```

wobei `Number` auch durch einen beliebigen Supertyp von `Number` ersetzt werden darf, jedoch durch keinen Subtyp. Das Einfügen von Elementen in `liste` bleibt jedoch weiter nicht gestattet, da nicht bekannt ist, welchen Typs die Elemente mindestens sein müssen.



Selbsttestaufgabe 43.1

Überprüfen Sie die letzte Aussage, indem Sie versuchen, ein Gegenbeispiel zu finden.

Per `extends` beschränkte Typ-Wildcards erlauben also eine spezielle Art des Subtypings, nämlich eine, bei der Zuweisungen von Objekten eines Subtyps ganz normal an Variablen eines Supertyps erlaubt sind, aber in der Folge schreibende Zugriffe auf Variablen, die mit dem Typparameter als Typ deklariert sind, verboten (lesende Zugriffe sind hingegen erlaubt). Die Situation unterscheidet sich von der bei den Arrays (Kapitel 41) lediglich dadurch, dass die Typprüfung statisch, also bereits zur Übersetzungszeit durchgeführt wird. Es ist damit sichergestellt, dass es niemals zu einem Laufzeitfehler entsprechend der `Array store exception` kommt; ein entsprechender dynamischer Typ-Test kann damit entfallen. Wie allgemein üblich werden damit auch Verwendungen ausgeschlossen, die eigentlich legal wären; so ist z. B.

spezielle Art des Subtypings

```
1075 List<? extends Integer> liste = new ArrayList<Integer>();  
1076 liste.add(new Integer(1));
```

nicht zulässig, obwohl hier eigentlich kein Problem vorliegt.

Nun kann man sich fragen, ob nicht auch eine umgekehrte, nur schreibenden Zugriff erlaubende Art des Subtyping möglich ist. Die Antwort ist einfach: ja. Man muss dazu nur die Beschränkung des Typ-Wildcards umkehren und verlangen, dass nur Supertypen der Schranke eingesetzt werden dürfen. Da Supertypen mit ihren Subtypen zuweisungskompatibel sind, weiß der Compiler, dass er Elemente jedes beliebigen Subtyps der Schranke zuweisen darf. Für eine Deklaration

nach unten beschränkte Typ-Wildcards

```
1077 List<? super Integer> liste;
```

beispielsweise, bei der `super Integer` die untere Schranke für tatsächliche Typparameter bei einer Zuweisung angibt, ist

```
1078 liste = new ArrayList<Number>();
```

und in der Folge sogar

```
1079 liste.add(1);
```

erlaubt, denn Listen mit Elementtyp `Number` (oder einem beliebigen anderen Supertyp von `Integer`) können, aufgrund des Subtypings der Elementtypen, problemlos `Integer` zugewiesen werden. Der lesende Zugriff auf Elemente von `liste` hat jedoch immer den Typ `Object` zum Ergebnis, so dass

```
1080 Integer i = liste.get(1)
```



bei obiger Deklaration von `liste` nicht erlaubt ist: Der Elementtyp dürfte ja auch ein Supertyp von `Integer` sein, so dass die Zuweisung zu `i` ungültig wäre. `List<? super Integer>` ist also spezieller Supertyp von allen Instanzen von `List<E>`, deren tatsächlicher Typparameter (also der für `E` eingesetzte Typ) ein Supertyp von `Integer` ist. Der Supertyp ist speziell, weil zwar eine Zuweisungskompatibilität gegeben ist, der Zugriff auf die Elementtypen (die mit dem Typparameter typisierten Elemente des Typs) auf schreibenden beschränkt ist.

Selbsttestaufgabe 43.2

Überlegen Sie, ob es möglich und sinnvoll ist, für einen Typparameter ein Typ-Wildcard mit oberer und unterer Schranke anzugeben.

Durch die mögliche Beschränkung von Typ-Wildcards entsteht für jeden generischen Typ eine (potentiell unendlich große) Menge von Typen, so dass man sich fragen kann, ob diese Typen in einer bestimmten Subtypbeziehung zueinander stehen. Dies ist tatsächlich der Fall: Für mit `extends` nach oben beschränkte Wildcard-Typen gilt, dass wenn die Schranken Subtypen sind, dann auch die Wildcard-Typen Subtypen sind. Wenn also `Integer` ein Subtyp von `Number` ist, dann ist auch `List<? extends Integer>` ein Subtyp von `List<? extends Number>`. Umgekehrt gilt für mit `super` nach unten beschränkte Typen, dass `List<? super Integer>` ein Supertyp von `List<? super Number>` ist. Man sagt auch, das Subtyping mit `extends` beschränkter Wildcard-Typen sei **kovariant** (da das Subtyping der Wildcard-Typen sich am Subtyping der Schranken orientiert) und das mit `super` beschränkter Wildcard-Typen **kontravariant** (aus entsprechendem Grund).

Subtyping von Wildcard-Typen

Selbsttestaufgabe 43.3

Überlegen Sie sich, ob `ArrayList<? extends Integer>` ein Subtyp von `List<? extends Number>` ist.

Ein typisches Beispiel für die Verwendung von Wildcard-Typen ist das folgende:

typisches Beispiel

```
1081 class List<T> {  
1082     copyFrom (List<? extends T> andereListe) {...}  
1083     copyTo (List<? super T> andereListe) {...}  
1084     ...  
1085 }
```



43.3 Beschränkter parametrischer Polymorphismus in JAVA

Wie bereits in Kurseinheit 3, Abschnitt 29.4 erwähnt, kauft man sich mit einfachem parametrischem Polymorphismus außerhalb der Typdefinition Typsicherheit zum Preis der mangelnden Typsicherheit innerhalb: Solange man keine Aussagen über den konkreten Typ, der für einen Typparameter eingesetzt wird, machen kann, kann man bei der Implementierung einer Klasse, die den parametrischen Typ definiert, auch keine Eigenschaften der Objekte, die von dem (unbekannten) Typ sein sollen, voraussetzen. Was man vielmehr braucht, ist beschränkter parametrischer Polymorphismus. Natürlich gibt es den auch in JAVA.

Wenn man beispielsweise die parametrische Definition sortierter Listen, `SortedList<E>`, auf Elementtypen einschränken will, die Subtypen von `Comparable` sind (damit die in `Comparable` definierte Methode `compareTo(.)` zur Verfügung steht), dann schreibt man in JAVA

```
1086 interface SortedList<E extends Comparable> {  
1087     void insert(E element);  
1088     void remove(E element);  
1089     ...  
1090 }
```

Die möglichen Werte der Typvariable `E` werden dadurch auf Typen eingeschränkt, die `Comparable` (direkt oder indirekt) erweitern. Die Implementierung der Methode zum Einfügen und Entfernen von Elementen in sortierten Listen kann also davon ausgehen, dass alle Objekte, die in einer solchen Liste gespeichert sind, die Nachricht `compareTo(.)` verstehen, die vom Interface `Comparable` vorgeschrieben wird. Man beachte, dass dadurch keine neuen Subtypenrelation zwischen irgendwelchen Typen hergestellt wird — es werden lediglich die Möglichkeiten, konkrete Typen (z. B. in Variablendeclarationen) zu bilden, eingeschränkt. Eine Variablendeclaration

```
1091 SortedList<String> liste;
```

wobei `String` ein Subtyp von `Comparable` ist, ist also möglich, eine wie

```
1092 SortedList<Point> liste;
```

hingegen nicht. Dennoch ist `SortedList<String>`, wie bereits in Abschnitt 29.3 bemerkt, kein Subtyp von `SortedList<Comparable>`.

Selbsttestaufgabe 43.4 (nur für JAVA-Programmiererinnen)

Probieren Sie aus, ob zwei Instanzen generischer Typen (also z. B. `ArrayList<Object>` und `ArrayList<String>` als Instanzen von `ArrayList<E>`) in JAVA zur Laufzeit denselben oder verschiedene Typen haben. Verwenden Sie dazu die Methode `getClass()` auf Objekten, die Sie von diesen Typen erzeugt haben.



43.4 Rekursiv beschränkter parametrischer Polymorphismus

Wenn Ihnen das noch nicht kompliziert genug ist, geht es noch weiter: Es ist nämlich `Comparable` selbst ein Typ, der von einer Parametrisierbarkeit profitieren würde. In einem ersten Ansatz würde man verlangen, dass von zwei vergleichbaren Elementen beide vom selben Typ sein müssen. Dies könnte man durch die Deklaration von

```
1093 interface Comparable<T> {  
1094     int compareTo(T o);  
1095 }
```

(tatsächliche Definition von `Comparable` in JAVA) sowie

```
1096 interface SortedList<E extends Comparable<E>> ...
```

erreichen. Man beachte die Parallelität zu STRONGTALK (Abschnitt 29.5).

Bei genauerer Betrachtung des Beispiels stellt sich allerdings heraus, dass die Bedingung, dass die zu vergleichenden Elemente für eine Sortierung alle vom selben Typ sein müssen, zu hart ist. Vielmehr würde es ja ausreichen, wenn die Objekte der sortierten Liste unter anderem mit den Objekten desselben Typs vergleichbar wären — sie könnten mit Objekten von noch mehr Typen vergleichbar sein. Die Bedingung muss also lauten, dass der Parametertyp von `Comparable` mindestens `E` umfassen muss, er kann aber auch allgemeiner, z. B. `Object` (für einen Vergleich beliebiger Objekte, wie auch immer der erfolgen soll), sein. Genau dies wird durch

Wildcards in Typdefinitionen

```
1097 interface SortedList<E extends Comparable<? super E>> ...
```

ausgedrückt, wobei das Typ-Wildcard wieder für einen beliebigen Typen, der die nachfolgende Bedingung erfüllt, steht. Im gegebenen Fall muss es sich bei dem für das Wildcard einzusetzenden Typ um einen (direkten oder indirekten) Supertypen von `E` handeln (wobei `E` ja selbst eine Typvariable ist, die erst bei einer Variablendeclaration an einen konkreten Typ gebunden wird). So ist

```
1098 SortedList<String> liste;
```

zulässig, wenn `String` wie in

```
1099 class String extends Comparable<String>
```

oder in

```
1100 class String extends Comparable<Object>
```

deklariert ist.



43.5 Generische Methoden

Außer in Typdefinitionen können Typvariablen (formale Typparameter) auch in Methodendefinition eingeführt (deklariert) werden. In diesem Fall ist die Sichtbarkeit der Typparameter auf die jeweilige Methode beschränkt.⁷² Es lassen sich damit variable Über- und Rückgabeparametertypen für eine Methode deklarieren.⁷³ Die Belegung der Typparameter mit konkreten Typen als Werten erfolgt dann bei der Bindung eines Methodenaufrufs zur Übersetzungszeit. Der tatsächliche Typparameter muss dabei nicht angegeben werden, wenn er sich aus dem Kontext eindeutig ergibt (sog. *Typinferenz*).

So handelt es sich beispielsweise bei der Deklaration

```
1101 public static <T> List<T> toList(T[] array);
```

wobei T nicht zugleich Typparameter der die Methodendefinition enthaltenden Klasse sein darf, um die Deklaration einer Methode, die ein Array eines beliebigen Typs, hier durch T repräsentiert, in eine Liste mit Elementen desselben Typs konvertiert.

Beim Aufruf einer parametrischen Methode muss der tatsächliche Typparameter angegeben werden. Allerdings erlaubt der Java-Compiler, diesen wegzulassen, wenn er sich aus dem Aufruf erschließen lässt (die oben erwähnte Typinferenz). Bei obiger Methodendeklaration ist das z. B. beim Aufruf

```
1102 Lists.toList(new Integer[] {1, 2, 3});
```

oder

```
1103 Lists.toList(new String[] {"abc", "def"});
```

der Fall: Hier wird der Typ `Integer` beziehungsweise `String` aus dem Typ des tatsächlichen Parameters abgeleitet. Dies ist jedoch nicht immer möglich; ein Aufruf wie

```
1104 Lists.<Integer>toList(new Integer[] {1, 2, 3});
```

macht den tatsächlichen Typparameter dann explizit.

Selbsttestaufgabe 43.5

Erläutern Sie, welchen Nachteil die konventionelle Definition von `toList(.)` mit der Signatur

```
1105 public static List toList(Object[] array);
```

⁷² Man bedenke, dass in JAVA durch eine parametrische Methodendefinition anders als durch Klassendefinitionen keine neuen Typen erzeugt werden.

⁷³ Man beachte, dass diese Variabilität nichts mit Ko- und Kontravarianz zu tun hat; vgl. hierzu auch Kurseinheit 3, Kapitel 30.



hat.

Der (formale) Typparameter einer generischen Methode kann zunächst an jeden beliebigen Typ gebunden werden. Genau wie bei generischen Klassen reduziert dies jedoch entweder die Verwendbarkeit oder die Typsicherheit der mit dem Parameter übergebenen Objekte, da deren Typ innerhalb der Methode unbekannt ist. Es ist also, wieder genau wie bei generischen Klassen, möglich, den Typparameter mit `extends` zu beschränken.

Beschränkung der Typparameter in generischen Methoden

Nicht selten wird der Typparameter (nicht die mit ihm deklarierten Parameter!) innerhalb der Methode nicht mehr verwendet. Eine Variable, die nur einmal vorkommt, kann man aber auch weg- oder zumindest unbenannt lassen. Und so ist es in diesen Fällen gute Praxis, anstelle des Typparameters ein (entsprechend beschränktes) Typ-Wildcard zu verwenden.

Typ-Wildcards in Methoden

43.6 Generische Variablen

Schließlich ist es in JAVA auch noch möglich, Variablen generisch zu deklarieren, also ohne dass die in der Variablen Deklaration verwendete Typvariable bereits von einer umschließenden Methode oder Klasse eingeführt worden wäre. Allerdings geht das, da eine Variable anders als eine Klasse oder Methoden keinen neuen Sichtbarkeitsbereich definiert, nur mit anonymen Typvariablen, also Wildcards. Man beachte jedoch, dass eine solche Variablen-deklaration, genau wie eine generische Methodendefinition, keinen neuen Typ einführt (s. Fußnote 72); vielmehr handelt es sich mit dem durch ein Typ-Wildcard als tatsächlichem Parameter eingesetzten Typ selbst um einen konkreten Typ. Der per

```
1106 List<?> liste;
```

deklarierten Variable `liste` können also Listen beliebigen Elementtyps zugewiesen werden, einfach weil `List<?>` (unter den in Abschnitt 43.2 genannten Einschränkungen) Supertyp aller Instanziierungen von `List<T>` ist.

44 Dynamische Typprüfung in JAVA

Wie Sie gesehen haben, hat das Typsystem JAVAs eine starke statische Komponente. Gleichwohl können nicht alle Typprüfungen zur Übersetzungzeit durchgeführt werden. Während man die dynamischen Typprüfungen bei Arrays (Kapitel 41) noch durch die Einführung von Wildcards hätte vermeiden können, gilt das nicht für die gelegentlich notwendigen *Typumwandlungen* (Type casts; zumindest nicht für alle).



44.1 Type casts

Wie bereits in Kapitel 27 von Kurseinheit 3 erläutert, versteht man unter einem Type cast (einer *Typumwandlung*) den Vorgang, bei dem einem Programmelement ein anderer Typ aufgezwungen wird als der, mit dem es deklariert wurde. Type casts werden also insbesondere auf Variablen und auf Methodenausdrücke angewendet, und zwar immer dann, wenn man diese einer Variable zuweisen will, zu der keine Zuweisungskompatibilität besteht, oder wenn man darauf eine Methode aufrufen (oder ein Feld zugreifen) will, das der deklarierte Typ nicht anbietet. So führt z. B.

```
1107 Object o = new String("abc");  
1108 o.substring(2, 3);  
1109 String s = o;
```

zu zwei Typfehlern: einem, weil `o` vom Typ `Object` ist, der keine Methode `substring` kennt, obwohl das Objekt, auf das `o` verweist, diese Methode sehr wohl kennt, und einem, weil `Object` nicht zuweisungskompatibel mit `String` ist. Zwei Type casts lösen das Problem:

```
1110 ((String) o).substring(2, 3);  
1111 String s = (String) o;
```

Wie schon in Kapitel 27, so unterscheiden wir auch hier in Abhängigkeit davon, wie die beteiligten Typen, der Ausgangstyp und der Zieltyp eines Type casts, miteinander in Beziehung stehen, drei Arten von Typumwandlungen:

drei Arten von Type casts

- *Up casts*: der Zieltyp ist ein Supertyp des Ausgangstyps
- *Down casts*: der Zieltyp ist ein Subtyp des Ausgangstyps
- *Cross casts*: Zieltyp und Ausgangstyp stehen in keiner Subtypenbeziehung zueinander, stehen also gewissermaßen nebeneinander.

Bei obigem Beispiel handelt es sich um (die bei weitem häufigsten) Down casts.

Wie man sich leicht vor Augen hält, ist ein Up cast auch in JAVA immer sicher, da jedes Objekt neben seinem Typ zugleich auch den Typ all seiner Supertypen hat (weswegen der Up cast auch immer weggelassen werden kann); Down und Cross casts sind jedoch nur erfolgreich, wenn das Objekt, zu dem der typumgewandelte Ausdruck auswertet, tatsächlich den Zieltyp (oder einen Subtyp davon) hat. Dies kann jedoch zur Übersetzungszeit nicht garantiert werden; Down und Cross casts können daher zu Laufzeittypfehlern (der in JAVA sog. Class cast exception) führen.

Mit der Einführung von Generics ergeben sich ein paar Probleme mit impliziten Typumwandlungen. Das nachfolgende Beispiel ist jedoch nur für Interessierte; insbesondere auf sog. *Raw types* und das damit zusammenhängende Konzept der *Type erasure* wird in diesem Kurs nämlich nicht eingegangen.

Type erasure



Bei Vorliegen der folgenden parametrisierten Klassendefinition

```
1112 class Kiste<T> {  
1113     T inhalt;  
1114     void reintun (T x) { inhalt = x; }  
1115     T rausnehmen () { return inhalt; }  
1116 }
```

und folgender Variablen Deklarationen und Initialisierungen

```
1117 Kiste kiste = new Kiste(); // sog. Raw type  
1118 Kiste<Tier> tierkiste = new Kiste<Tier>();
```

ergeben die Zuweisungen

```
1119 kiste = tierkiste; // geht ohne Warnung  
1120 kiste.reintun(new Object()); // Warning:  
1121     // Type safety: The method reintun(Object)  
1122     // belongs to the raw type Kiste. References to generic  
1123     // type Kiste<T> should be parameterized  
1124 Tier tier = tierkiste.rausnehmen(); // geht ohne Warnung,  
1125     // aber ClassCastException!!!
```

bei der letzten einen Laufzeitfehler vom Typ Class cast exception.

44.2 Typtests

Laufzeitfehler der obigen Art vermeidet man in JAVA mit Hilfe sogenannter Typtests, die man vor einer Typumwandlung durchführt. Das dazugehörige Schlüsselwort heißt `instanceof`:

```
1126 if (o instanceof String)  
1127     ((String) o).substring(2, 3);  
1128 if (o instanceof String)  
1129     String s = (String) o;
```

bilden die typsichere Variante der Typumwandlung. Es bleibt allerdings an der Programmiererin hängen, zu sagen, was passieren soll, wenn der Wert von `o` nicht den geforderten Typ (hier `String`) hat.

45 Programmieren mit Interfaces

Oben hatten wir ja bereits erwähnt, dass eine Klasse neben den in ihrem Interface veröffentlichten noch weitere öffentliche (`public` deklarierte) Methoden haben kann. Daraus folgt, dass unterschiedliche Interfaces einer Klasse unterschiedliche Methodenmengen zur Verfügung stellen können. Da auf einer

Interfaces bieten
klientenspezifische
Sichten

Variable, die mit einem Interfacetyp deklarierte wurde, aber immer nur die Methoden aufgerufen werden dürfen, die der Interfacetyp veröffentlicht (sonst könnte aus der erfolgreichen Prüfung der Zuweisungskompatibilität eines Programms nicht abgeleitet werden, dass auch keine Laufzeitfehler der Art, dass eine Methode nicht vorhanden ist, auftreten können), können mit Hilfe von verschiedenen Interfaces verschiedene Sichten auf ein Objekt gewährt werden. So kann beispielsweise ein Ein- und Ausgabestrom als Klasse zwei Interfaces implementieren, von denen das eine nur die zum Lesen benötigten Methoden enthält und das andere nur die zum Schreiben:

```
1130 interface ReadStream<T> {
1131     T read();
1132 }
1133 interface WriteStream<T> {
1134     void write(T object);
1135 }
1136 class IOStream<T> implements ReadStream<T>, WriteStream<T> {...}
```

Verschiedene Klienten von Instanzen dieser Klasse könnten dann jeweils entweder nur lesend oder nur schreibend auf einen Ein-/Ausgabestrom zugreifen, und zwar abhängig davon, welches Interface die Variable, die auf den Strom verweist, zum Typ hat:

Beschränkung des Zugriffs

```
1137 class WillNurLesen {
1138     ReadStream<String> eingabe = new IOStream<String>();
1139
1140     void tueEtwas() {
1141         eingabe.read(); // OK
1142         eingabe.write("man kann's ja mal versuchen!"); // Syntaxfehler
1143     }
1144
1145 class WillNurSchreiben {
1146     WriteStream<String> ausgabe = new IOStream<String>();
1147
1148     void tueEtwas() {
1149         eingabe.write("so ist's recht"); // OK
1150         eingabe.read(); // Syntaxfehler
1151     }
1152 }
```

Umgangen werden kann diese Zugriffsbeschränkung über Interfaces in JAVA immer dann, wenn die Klasse, von dem das verwendete Objekt eine Instanz ist, ebenfalls als Typ zur Verfügung steht — man kann in Variablendeklarationen dann genauso gut die Klasse als Typ verwenden. So könnte man sowohl in **WillNurLesen** die Variable **eingabe** als auch in **WillNurSchreiben** die Variable **ausgabe** mit dem Typ **IOStream** deklarieren, hätte damit jedoch keine syntaktische Unterstützung (Einschränkung des Funktionsumfangs) bei der Benutzung der Variable mehr. C# geht hier einen Schritt weiter und ermöglicht Klassendefinitionen zumindest, ihre Me-

die Verwendung von Interfaces in JAVA ist freiwillig

thoden ausschließlich über Interfacetypen zugänglich zu machen (s. Kurseinheit 5, Abschnitt 50.4.2). Seit JAVA 8 kann man allerdings in Interfaces auch statische Methoden definieren, die neue Instanzen von Klassen, die die Interfaces implementieren, zurückliefern und somit die Konstruktoren dieser Klassen ersetzen. Man braucht dann keinerlei Zugriff mehr auf die Klassen, die deswegen als *package local* deklariert und die somit vollständig hinter dem Interface als Schnittstelle verborgen werden können.

Interfaces dienen aber nicht nur der Beschränkung des Zugriffs wie im obigen Beispiel — sie dienen auch der besseren Austauschbarkeit von Klassen. Und das bringt uns zurück zum eingangs Kapitel 43 benutzten Beispiel (Zeile 1051 ff.): Dadurch, dass die Variable **liste** mit dem Typ **List** (ein Interface) und nicht **ArrayList** (eine Klasse) deklariert wurde, können ihr auch Instanzen anderer Klassen als **ArrayList** zugewiesen werden, solange sie nur dasselbe Interface implementieren. Das Interface als Typ der Variable verlangt lediglich, dass alle in ihm versammelten Methoden von den Objekten, auf die die Variable per Typsystem verweisen darf, auch angeboten werden, und das wird dadurch sichergestellt, dass der Typ der rechten Seite aller Zuweisungen zur Variablen konform zum Typ der linken Seite sein muss. Die Zuweisungskompatibilität ist in JAVA also an die Typkonformität gebunden, und zwar an die nominale.

Interfaces erhöhen Austauschbarkeit

Eine weitere, vergleichsweise häufig verwendete Möglichkeit des Einsatzes von Interfaces ist die als sog. **Tagging** oder **Marker interfaces**. Ein Tagging oder Marker interface hat in der Regel keine eigenen Methodendeklarationen, sondern dient lediglich der *Filterung von Variablenwerten*.

Tagging oder Marker interfaces

1151 `interface Markiert {}`

wäre ein solches Interface. Da in JAVA die vom Compiler statisch geprüfte Zuweisungskompatibilität ja Namenskonformität verlangt, kann die Typisierung einer Variable mit dem Interface **Markiert** erzwingen, dass dieser Variable nur Instanzen solcher Klassen zugewiesen werden, die das Interface **Markiert** zu implementieren deklarieren. Da das Interface aber keine Auflagen macht (keine Methodendeklarationen vorgibt, die von der Klasse mit Definitionen versehen werden müssen), ist die Implementierung des Interfaces für die Klasse zunächst ohne Konsequenzen. Die durch das Interface erfolgte Markierung der Klassen (bzw. deren Instanzen) kann jedoch zur Übersetzungszeit vom Compiler (s. o.) und zur Laufzeit durch einen Typtest (s. Abschnitt 44.2) überprüft werden.

Ein konkretes Beispiel für ein Marker interface in JAVA ist das Interface **Serializable**, mit dessen Implementierung eine Klasse deklariert, dass ihre Instanzen serialisiert werden können. Die Serialisierung wird aber nicht von der Klasse selbst vorgenommen, sondern von einer anderen; die Klasse muss also zu ihrer Serialisierung nichts weiter beitragen. Seit JAVA 5 würde man allerdings solche Marker-Interfaces eher durch Metadaten (auch *Annotationen* genannt; s. Abschnitt 47.4) ersetzen, so wie das in C# schon länger gehandhabt wird (s. Abschnitt 50.4 in Kurseinheit 5).

Beispiel Serializable



JAVAs Interface-als-Typ-Konzept ist ziemlich interessant und vielleicht sogar der größte einzelne Beitrag JAVAs zur objektorientierten Programmierung. In der Programmierpraxis scheint es jedoch, sieht man einmal von großen Frameworks ab, nur langsam anzukommen. Das mag zum einen an der schlechten Verkaufsstrategie liegen („Ersatz für Mehrfachvererbung“ — jede Anfängerin merkt nach fünf Minuten, dass das nicht stimmt), zum anderen aber auch an der Vergrößerung des Programmumfangs, die der parallelen Definition von Klassen und Interfaces geschuldet ist (die auch den Wartungsaufwand erhöhen kann, obwohl ja Interfaces eigentlich die Wartung vereinfachen sollen). Für so manche Programmiererin scheinen die Möglichkeiten, die Schnittstelle einer Klasse mittels der Zugriffsmodifikatoren innerhalb der Klasse selbst zu definieren, völlig auszureichen. Der Preis dafür ist eine mangelnde Differenzierbarkeit des Zugriffs nach verschiedenen Klienten sowie eine (häufig vorschnelle) Festlegung von Variablen auf Instanzen einer Klasse. Mehr zur sog. interfacebasierten Programmierung finden Sie im Kurs 01853.



Kurs

46 Interne und externe Iteration über Collections

Genau wie in der SMALLTALK-Programmierung kommt man in der JAVA-Programmierung häufig in die Verlegenheit, *:n*-Beziehungen umsetzen zu müssen. Wie in SMALLTALK geschieht dies auch in JAVA mit Hilfe von *Zwischenobjekten*. Anders als in SMALLTALK wird hier aber zwischen zwei Arten von Zwischenobjekten grundsätzlich unterschieden: den Arrays und den Collections.⁷⁴ Da JAVA-Arrays in ihrer Funktionalität beschränkt sind (s. Kapitel 41: keine eigenen Methoden zur Unterstützung des Zugriffs, kein dynamisches Wachstum, dazu die etwas verkorkste Situation beim Subtyping), werden Arrays vornehmlich dort eingesetzt, wo es um Effizienz (geringer Speicherverbrauch und schneller Zugriff) geht.



46.1 Externe Iteration

Bei *:n*-Beziehungen müssen ja häufig mehrere Elemente der gleichen Behandlung unterzogen werden oder es werden aus der Menge der Elemente einige gesucht. Sind die Elemente in einem Array gespeichert, so kann man in JAVA dazu etwas der Form

```
1152 for (int i = 0; i < a.length; i = i + 1) {... a[i] ...}
```

schreiben. Verwendet man aber Collections, die nicht indiziert sind (es gibt auch in JAVA so Collections wie Bäume oder verkettete Listen), ist die obige Form der Iteration nicht einsetzbar. Stattdessen gibt es in JAVA die beiden Interfaces **Enumeration** (total veraltet) und **Iterator** (etwas neuer), die eine einheitliche Iteration über Collections mittels sog. *Iteratoren* erlauben. Jede Collection-Klasse, die **Iterator** implementiert, bietet dazu eine Methode **iterator()** an, die ein solches Iteratorobjekt (vom Typ **Iterator**) zurück liefert. Mit der Methode **next()** erhält man von diesem Objekt zunächst das erste und in der Folge

⁷⁴ Zur Erinnerung: In SMALLTALK sind Arrays spezielle Collections, mit eigenen Methoden.



alle weiteren Objekte der Collection; mit der Methode `hasNext()` fragt man ab, ob die Collection noch weitere Objekte enthält. Eine typische Iteration über eine Collection sah in JAVA also wie folgt aus:

```
1153 for (Iterator i = aCollection.iterator(); i.hasNext();)  
1154     {... i.next() ...}
```

Alternativ (und in der Praxis nicht seltener, obwohl eine Anweisung mehr) findet man auch

```
1155 Iterator i = aCollection.iterator();  
1156 while (i.hasNext()) {... i.next() ...}
```

In manchen Programmen findet man allerdings auch heute noch die vor JAVA 2 einzig mögliche Variante

```
1157 Enumeration e = aCollection.elements();  
1158 while (e.hasMoreElements()) {... e.nextElement() ...}
```

bzw. das Äquivalent mit der For-Schleife.

Die Wahl der unnötig langen Bezeichner lässt vermuten, dass man die Häufigkeit solcher Konstruktionen in der Programmierpraxis unterschätzt hatte. Dieser Fehler wurde mit JAVA 5 korrigiert, so dass man seither stattdessen

For-each-Schleife

```
1159 for (Element e : aCollection) {... e ...}
```

schreiben kann (wobei `Element` hier für den Elementtyp von `aCollection` steht). Voraussetzung ist allerdings, dass die Klasse von `aCollection` das Interface `Iterable` implementiert, das wiederum verspricht, dass die Instanzen die Methode `Iterator iterator()` anbieten und damit einen Iterator zurückliefern, der dann von der For-Anweisung zur Produktion der Elemente ausgenutzt werden kann; es handelt sich bei dieser Version der For-Schleife also lediglich um „syntaktischen Zucker“. Man beachte, dass dabei eine weitere Verquickung von Sprachdefinition und Klassenbibliothek (API; speziell der Interface `Iterator`) stattgefunden hat, die nicht schön ist.

46.2 Interne Iteration

Von SMALLTALK kennen Sie ja schon die Möglichkeit, Iterationen als Methoden der Collection-Klassen zu definieren (die *interne Iteration*; s. Abschnitt 4.6.4 in Kurseinheit 1 und Kapitel 13 in Kurseinheit 2). Mit Version 8 wurden nun auch in JAVA (anonyme) Funktionen (die sog. *Lambda-Ausdrücke*) eingeführt, die interne Iterationen à la SMALLTALK erlauben. Allerdings wurden für die Implementierung der Methoden `forEach`, `filter`, `map` und `reduce` (entsprechend SMALLTALKS `do`, `select`, `collect` und `inject`) ein eigenes Framework geschaffen: die sog. *Streams*.



Streams sind ein mit JAVA 8 neu eingeführtes Konstrukt, das eine Datenverarbeitung in Pipelines ganz ähnlich wie die Collections in SMALLTALK erlaubt: Das Ergebnis einer Operation wie `filter` oder `map`, auf einem Stream ausgeführt, ist wieder ein Stream, auf dem weitere Operationen dieser Art ausgeführt werden können (das *Pipelining*). Das besondere an Streams ist, dass sie keine Datenspeicher wie Collections sind, sondern *interne Iteratoren*: Jeder Stream für sich hält zu einem Zeitpunkt immer genau ein Element einer potentiell unendlichen Folge von Elementen. Dabei dienen Streams entweder abgeschlossene Datenspeicher wie Collections und Files oder offene Datenlieferanten wie die Tastatur oder Datengeneratoren (wie beispielsweise ein Zufallszahlengenerator oder eine Vorschrift zur Berechnung einer Fibonacci-Folge) als Quelle. Ein Ergebnis liefert eine Stream-Pipeline (die auch aus nur einem Element bestehen kann) immer erst durch einen sog. Abschluss, also eine Methode wie `reduce` oder `collect`, die ein anderes Ergebnis als einen Stream liefert (eine Collection im Falle von `collect`).

Das Stream-Framework von JAVA ermöglicht für viele Pipelines eine besonders effiziente Ausführung. So muss jedes Element einer Datenquelle in der Regel nur einmal angefragt werden und die interne Speicherung von Zwischenergebnissen wird automatisch gering gehalten. Insbesondere wird gegenüber einer naiven Implementierung von Pipelines, die eine Iteration je Element der Pipeline benötigt, in vielen Fällen nur eine Iteration gebraucht (die sog. Stream fusion). Zudem kann durch Wahl des Streams (und nicht der Operationen) bestimmt werden, ob die Verarbeitung der Pipeline parallel oder sequentiell erfolgt. Gleichwohl ist die Performanz von JAVA-Pipelines nicht leicht vorherzusagen und eine gewinnbringende Verwendung setzt sicherlich einiges an Erfahrung und Kenntnis der Implementierung voraus.



47 Spezielle Klassen

Grundsätzlich sollte eine Klassenbibliothek mit den Mitteln der Sprache programmiert, ansonsten aber von der Sprachdefinition unabhängig und damit austauschbar sein. Dies gilt natürlich genauso für die Prozeduren in imperativen Sprachen: Auch diese sollten nicht Teil der Sprache, sondern lediglich darin geschrieben werden. Nun hat man sich aber schon bei den edelsten imperativen Sprachen nicht daran gehalten (in PASCAL beispielsweise sind `read` und `write` Bestandteil der Sprache und der Compiler weiß, dass ihnen — als einzigen Prozeduren — beliebig viele Parameter übergeben werden dürfen, wobei jeder einzelne Parameter in seinem Typ nicht festgelegt ist); bei den objektorientierten tut man es erst recht nicht. So sind in JAVA einige Klassen mit der Sprache fest vorgegeben und können nicht beliebig ersetzt werden.



47.1 Object

Die Klasse `Object` gibt die Eigenschaften vor, die allen Objekten, einschließlich Array-Objekten, gemeinsam sind. Darunter sind keine Felder (Instanzvariablen), aber elf Methoden. Es sind dies:

- `Object clone()` zum Erzeugen von Kopien (vergleichbar SMALLTALKS `copy`)
- `boolean equals(Object)` zum Test auf Gleichheit anstelle von Identität (entsprechend SMALLTALKS `=`)
- `String toString()`, die eine String-Präsentation des Objekts zurückgibt (entsprechend SMALLTALKS `printString`)
- `Class<? extends Object> getClass()` mit dem offensichtlichen Zweck (entsprechend SMALLTALKS `class`)
- `int hashCode()` für die Speicherung von Objekten in Hash-Tabellen (entsprechend SMALLTALKS `hash`)
- `void finalize()`, die vom Garbage collector aufgerufen wird, wenn das Objekt aus dem Speicher entfernt wird (es können damit externe Ressourcen, die mit dem Objekt verbunden sind, freigegeben werden, also z. B. Dateien geschlossen)
- `void notify()` und `void notifyAll()` zur Benachrichtigung von Threads, die am Monitor des Objekt warten
- `void wait()` in drei Versionen, um den ausführenden Thread zum Warten zu bringen, entweder bis dem Objekt ein Notify gesendet wird oder ein anderes Ereignis eintritt.

In JAVA ist jeder Typ Subtyp von dem von `Object`. Das gilt auch für Interfaces. Man beachte aber, dass Interfaces ansonsten nicht von Klassen ableiten können.

47.2 Exception handling

Wie Sie schon bei den Anweisungen in Kapitel 38 gesehen haben, sieht JAVA ein sog. Exception handling vor. Beim Exception handling handelt es sich um eine Möglichkeit, bei der Spezifikation des Kontrollflusses eines Programms zunächst mögliche Ausnahmesituationen und deren Behandlung unberücksichtigt zu lassen und so zu tun, als würde immer alles gutgehen. Da das normalerweise auch der Fall sein sollte, erlaubt es sowohl der Autorin als auch der Leserin eines Programms, bei seiner hauptsächlichen Funktion zu bleiben, ganz nach dem Motto „zu den Ausnahmen kommen wir später!“

Die möglichen Ausnahmesituationen eines Programms werden in JAVA in Klassen eingeteilt, deren Instanzen jeweils eine konkrete Ausnahmesituation während des Programmablaufs repräsentieren. Wenn also beispielsweise in einem Programm auf ein Element eines

Exception-Klassen
und Try-catch-Blöcke

Arrays zugegriffen



werden soll, dessen Index außerhalb der Grenzen des Arrays liegt, dann erzeugt die JVM, die über die Einhaltung der Array-Grenzen wacht, eine Instanz der Klasse **ArrayIndexOutOfBoundsException** und wirft diese. Dabei bedeutet das Werfen einer Exception (genauer: einer Instanz einer Exception-Klasse), dass der Programmablauf an der gegenwärtigen Stelle abgebrochen und an der nächsten Stelle fortgesetzt wird, die angibt, den zum Typ (zur Klasse) der Exception-Instanz passenden Exception handler zu haben. Dazu ist es notwendig, die Codestrecke, in der die Exception auftreten kann, mit einer Try-catch-Klammer zu versehen, also mit einem Konstrukt der Form

```
1160 try {  
1161     ... anArray[anIndex] ...  
1162 }  
1163 catch (ArrayIndexOutOfBoundsException e) {...}  
1164 catch (<andere Exceptionklasse> e) {...}  
1165 ...
```

wobei der Typ der geworfenen Exception idealerweise in einer der Catch-Klauseln vor kommt. Ist dies nicht der Fall, gilt die Exception als durch die Try-catch-Klammer nicht behandelt und es wird die nächste umschließende Klammer gesucht (wobei mit nächste die vorherige im Programmablauf gemeint ist). Wird auf diese Weise keine passende Klausel gefunden, wird das Programm mit einer entsprechenden Fehlermeldung („Exception in thread ...“) abgebrochen.

Ausnahmesituationen können aber nicht nur durch die JVM, sondern auch durch das Programm selbst entdeckt und gemeldet (entsprechende Exceptions geworfen) werden. Dazu dient die schon erwähnte Throw-Anweisung, die mittels einer Instantiierung der Exception-Klasse eine entsprechende Ausnahme wirft. So verlagert beispielsweise

Throw-Anweisung

```
1166 try {  
1167     ...  
1168     if (anIndex < 0 || anIndex >= anArray.length)  
1169         throw new ArrayIndexOutOfBoundsException();  
1170     else {  
1171         ... anArray[anIndex] ...  
1172     }  
1173 }
```

den Test von der virtuellen Maschine ins Programm.

Try-catch-Anweisungen stehen wie fast alle Anweisungen in JAVA im Rumpf von Methoden. Wenn eine Methode eine bestimmte Exception wirft, ohne sie selbst zu fangen (also wenn die Throw-Anweisung nicht lokal von einer Try-catch-Anweisung umschlossen wird, die eine Catch-Klausel mit dem entsprechenden Typ enthält), dann verlangt die JAVA-Sprachdefinition zunächst, dass die Methode dies deklariert, dass sie also bekannt gibt, dass sie u. U. eine Exception des genannten Typs werfen wird. Dies geschieht mittels einer Throws-Klausel und sieht dann wie folgt aus:

Checked exceptions

```
1174 ... <Methodename> (<Parameterliste>) throws <Exceptionname> {...}
```



Eine Methode, die diese Methode aufruft, muss also den Aufruf entweder mit einer Try-catch-Anweisung klammern, die die geforderte Catch-Klausel enthält, oder selbst deklarieren, die Exception zu werfen. Diese Praxis führt zwar zu erheblicher Schreibarbeit, stellt aber letztlich die einzige Möglichkeit dar, zu erzwingen, dass sich die Programmiererinnen der möglichen Ausnahmesituationen, die auftreten können, bewusst sind, ohne die Spezifikation des Kontrollflusses dadurch über Gebühr zu belasten. Jede, die schon einmal in C die Aufrufe von Betriebssystemroutinen durch Abfrage der Return-Codes abzusichern versucht hat, weiß, wovon ich spreche. Man nennt Exceptions, die ein Auffangen innerhalb einer Methode oder eine Deklaration im Methodenkopf verlangen, **Checked exceptions**.

Nun gibt es aber Exceptions, die so gut wie überall auftreten können. Das prominenteste Beispiel ist vielleicht die Out of memory exception, die auftritt, wenn eine Speicheranforderung des Programms von der JVM nicht bedient werden kann. In der Praxis häufiger, wenn auch durch das Programm selbst vermeidbar, ist die Null pointer exception, die immer auftritt, wenn ein Ausdruck, auf dem ein Feld oder eine Methode zugegriffen werden soll, zu `null` ausgewertet wird (vgl. Selbsttestaufgabe 11.1). In dieselbe Kategorie fällt auch die Array index out of bounds exception, die sich natürlich durch vorsichtige Programmierung vermeiden ließe, die aber in der Praxis trotzdem immer wieder vorkommt. In all diesen Fällen wäre es außerordentlich mühsam, wenn man alle Anweisungen, bei denen die jeweilige Exception auftreten könnte, mit einer entsprechenden Try-catch-Anweisung absichern müsste.

Deswegen gibt es in JAVA Exceptions, bei denen das nicht nötig ist, die sogenannten **Unchecked exceptions**. Man könnte meinen, dass ob eine

Unchecked exceptions

Exception checked oder unchecked ist, von Fall zu Fall (von Auftreten zu Auftreten) von der Programmiererin, die sie wirft, zu unterscheiden wäre — dies ist aber nicht so. Stattdessen sind alle Exceptions, die von der Klasse `RuntimeException` abgeleitet sind, per Definition unchecked. `RuntimeException` ist selbst Subklasse der Klasse `Exception`, die wiederum Subklasse von `Throwable` ist. `Throwable` ist die Superklasse aller Klassen, die in einer Throw-Anweisung und in Catch-Zweigen bzw. Throws-Klauseln vorkommen dürfen. Mit `RuntimeException` wird also ein Zweig der Exception-Klassenhierarchie eingeleitet, dessen Elemente alle unchecked sind.

Neben `Exception` ist auch `Error` Subklasse von `Throwable`. Errors jedoch nicht gefangen werden, sondern zu einem sofortigen Programmabbruch führen. Die Konstruktion

Errors

```
1175 if (...) throw new Error("das ist schiefgegangen!");
```

wobei die Auslassungszeichen für die Formulierung einer Invariante stehen, hat man früher zur Emulation der inzwischen vorhandenen Assert-Anweisung verwendet; auch heute sollte man Errors eigentlich nur während der Testphase eines Programms einsetzen. Genau wie Exceptions der Sorte `RuntimeException` sind Errors unchecked.



47.3 Multi-threading

Ähnlich wie SMALLTALK erlaubt auch JAVA, parallele Ausführungsstränge zu programmieren. Diese heißen in JAVA jedoch nicht Prozesse, sondern Threads. Threads sind im Gegensatz zu den Prozessen eines Betriebssystems leichtgewichtig, was soviel heißen soll wie dass sich Threads die getrennte Allokierung von Ressourcen (wie Hauptspeicher) sparen, und alle auf denselben Ressourcen operieren. Es können also mehrere Threads innerhalb eines Prozesses laufen. Der Preis dafür ist, dass die Mechanismen zur Synchronisation bei Threads selbst realisiert werden müssen; bei Prozessen sind sie über die Inter-Prozess-Kommunikation des Betriebssystems geregelt. (Bei den Prozessen SMALLTALKS handelt es sich also auch eher um Threads als um Prozesse im eben beschriebenen Sinn.)



WIKIPEDIA

In JAVA wird ein neuer Thread gestartet, indem man eine neue Instanz der Klasse **Thread** erzeugt und auf ihr die Methode **start()** aufruft.

Thread erzeugen und starten

1176 `(new Thread()).start()`

startet also einen neuen Thread. Bleibt die Frage, was dieser Thread tut.

Die Klasse **Thread** besitzt dafür eine Methode **run()**, die von **start()** aufgerufen wird. Diese Methode ist jedoch leer, so dass der Thread gleich wieder beendet wird. Damit ein neuer Thread etwas Sinnvolles tut, gibt es zwei Möglichkeiten:

1. Man definiert eine neue Subklasse von **Thread** und überschreibt darin die Methode **run()** so, dass sie das Gewünschte tut oder zumindest anstößt.
2. Man lässt eine Klasse das Interface **Runnable** implementieren, implementiert dann in der Klasse die vom Interface geforderte Methode **run()**, erzeugt von dieser Klasse eine Instanz **i** und startet deren Methode **run()** mittels `(new Thread(i)).start()` (`start()` ruft dann `run()` auf **i** auf).

Auch im zweiten Fall wird eine Instanz der Klasse **Thread** erzeugt, die den neuen Thread repräsentiert. Man beachte jedoch, dass diese Instanz nicht selbst der Thread ist — der Thread ist, wie gesagt, ein paralleler Ausführungsstrang der JVM, der, genau wie der Ausführungsstrang, mit dem das Programm startet, nicht an ein Objekt gebunden ist, sondern mit dem Kontrollfluss zwischen den Empfängerobjekten hin- und herwechselt. Aktive Objekte, also Objekte, die ihren eigenen Ausführungsstrang haben und auch behalten (s. Kapitel 16), müssen in JAVA genau wie in SMALLTALK simuliert werden. Jeder Thread hat aber ein Thread-Objekt, das ihn repräsentiert; es kann mit **Thread.currentThread()** erfragt werden. Mit ihm sind so spezifische Daten wie der Name des Threads, seine Priorität etc. gespeichert.

Die Threads JAVAS benötigen also eine explizite Synchronisation. Ähnlich wie die Prozesse SMALLTALKS funktioniert dies mit Semaphoren, die hier allerdings **Monitore** genannt werden. Jeder Monitor ist mit einem Objekt verbunden (und jedes Objekt mit einem Monitor); wenn ein Thread einen Monitor eines Objektes sperrt

Synchronisation über Locking: Monitore



(„lockt“), dann kann kein anderer Thread den Monitor sperren, bevor die Sperre durch den ersten Thread wieder aufgehoben wird.

Es gibt zwei Möglichkeiten, die Synchronisation von Threads zu erzwingen. Die eine erfolgt mittels der `Synchronized`-Anweisung, die Sie oben schon kurz kennengelernt haben: Die Anweisungen eines Blocks können nur ausgeführt werden, wenn sie nicht gerade von einem anderen Thread ausgeführt werden. Das mit dem Block assoziierte Objekt, auf dessen Monitor die Sperre durchgeführt wird, wird explizit mit der `Synchronized`-Anweisung angegeben (häufig ist es `this`, also das Objekt, in dessen Kontext sich der Block befindet).

Möglichkeiten zur Synchronisation von Threads

Die zweite Möglichkeit ist, eine ganze Methode mit `synchronized` zu deklarieren. Wenn es sich dabei um eine Instanzmethode handelt, wird die Sperre auf dem Objekt, auf dem die Methode aufgerufen wurde, erwirkt; handelt es sich dagegen um eine Klassenmethode (also um eine, die `static` deklariert wurde), dann geht die Sperre auf das Objekt, das die Klasse repräsentiert. Felder lassen sich übrigens nicht `synchronized` deklarieren

47.4 Metaprogrammierung

Obwohl in JAVA längst nicht alles ein Objekt ist, gibt es doch die Möglichkeit, auf die Elemente eines Programms zuzugreifen. Dafür ist das sog. *Reflection API* zuständig, in dem für jede Art von Programmelement eine Klasse existiert, deren Instanzen entsprechende Programmelemente repräsentieren. (So gibt es dort eine Klasse `Method`, eine Klasse `Field` etc.) Eine genauere Betrachtung dieser API würde an dieser Stelle jedoch zu weit führen; sie wird im Kurs 01853 („Moderne Programmietechniken und -methoden“) ausführlicher behandelt.



Eine andere Form der Metaprogrammierung, die sog. **Annotationen**, haben zunächst nichts mit Objektorientierung zu tun — es werden damit lediglich Programmelementen im Quelltext Daten, sog. Metadaten, zugeordnet. Diese können dann während der Übersetzung und/oder während der Ausführung des Programms abgefragt werden und so den jeweiligen Prozess beeinflussen oder sogar steuern. In JAVA 5 wurden Annotationen als eine spezielle Art von Interfaces eingeführt, die jedoch keine Methoden, sondern nur Felder (!) deklarieren. Annotation sind ebenfalls Gegenstand des Kurses 01853.

Annotationen



48 Ein abschließendes Beispiel

Um Ihnen eine grobe Vorstellung davon zu geben, wie JAVA-Programme aussehen, finden Sie nachfolgend den Quellcode für ein Programm, das ein einfaches Ratespiel umsetzt. Das Programm besteht aus vier Klassen, nämlich der Klasse `Ratespiel`, die im wesentlichen die Startmethode nebst Initialisierung der Datenstruktur enthält, sowie den Klassen `Knoten`, `Tier` und `Merkmel`. Das Interface `Frage` dient der gemeinsamen Abstraktion von `Tier`



und Merkmal und verlangt von den beiden Klassen lediglich, dass sie eine Methode `stellen()` implementieren und auf den Aufruf derselben einen Wahrheitswert (die Antwort auf die gestellte Frage) zurückliefern. Die Methoden der Klasse `KeyboardInput` dienen der Interaktion mit der Benutzerin über die Konsole, deren Möglichkeiten in JAVA (wie in SMALLTALK) von Haus aus nur schwach ausgeprägt sind. `System.out` bezeichnet denn auch den *Ausgabestrom*, der mit der Konsole verbunden ist, und die Methode `println(.)` gibt etwas darauf aus.

```
1177 public class Ratespiel {  
1178     public static void main(String[] args) {  
1179         Knoten startKnoten = new Knoten(new Tier("Huhn"));  
1180         do {  
1181             System.out.println("Denke dir ein Tier aus und drücke Rtn!");  
1182             KeyboardInput.tasteDrücken();  
1183             startKnoten.fragen();  
1184             System.out.println("Möchtest du nochmal?");  
1185         } while (KeyboardInput.ja());  
1186     }  
1187 }  
  
1188 interface Frage {  
1189     boolean stellen();  
1190 }  
  
1191 public class Knoten {  
1192     Frage frage;  
1193     Knoten ja;  
1194     Knoten nein;  
  
1195     public Knoten(Frage frage) {  
1196         this.frage = frage;  
1197         nein = null;  
1198         ja = null;  
1199     }  
  
1200     public void fragen() {  
1201         boolean antwort = frage.stellen();  
1202         if (antwort) {// Antwort positiv  
1203             if (ja != null) // weitere Fragen vorhanden:  
1204                 ja.fragen(); // mit Ja-Knoten weitermachen  
1205             else // keine weiteren Fragen vorhanden:  
1206                 System.out.println("Fertig.");  
1207         }  
1208         else { // Antwort negativ  
1209             if (nein != null) // weitere Fragen vorhanden:  
1210                 nein.fragen(); // mit Nein-Knoten weitermachen  
1211             else { // keine weiteren Fragen vorhanden:  
1212                 String name = KeyboardInput.antwort("Wie heißt es dann?");  
1213                 Tier neuesTier = new Tier(name);  
1214                 String merkmalsfrage =  
1215                     KeyboardInput.antwort("Nenne mir eine Frage, die für "  
1216                     + neuesTier.name() + " mit Ja und für "  
1217                     + ((Tier) frage).name()  
1218                     + " mit Nein beantwortet werden muss!");  
1219                 Merkmal neuesMerkmal = new Merkmal(merkmalsfrage);  
1220                 ja = new Knoten(neuesTier);  
1221             }  
1222         }  
1223     }  
1224 }
```



```

1221         nein = new Knoten(frage); // frage wandert nach unten!
1222         frage = neuesMerkmal;
1223         System.out.println("Gut!");
1224     }
1225 }
1226 }
1227 }

1228 public class Tier implements Frage {
1229     private String name;

1230     public Tier(String name) {
1231         this.name = name;
1232     }

1233     public String name() {
1234         return name;
1235     }

1236     public boolean stellen() {
1237         System.out.println("Heißt es " + name + "?");
1238         return KeyboardInput.ja();
1239     }
1240 }

1241 public class Merkmal implements Frage {
1242     String frage;

1243     public Merkmal(String frage) {
1244         this.frage = frage;
1245     }

1246     public boolean stellen() {
1247         System.out.println(frage);
1248         return KeyboardInput.ja();
1249     }
1250 }

```

Die Klassen **Tier** und **Merkmal** sind beide recht klein und unterscheiden sich nur wenig. Beide speichern pro Instanz einen String, einmal in der Instanzvariable **name**, einmal in der Instanzvariable **frage**. Diese Strings werden jeweils bei der Erzeugung der Objekte per Konstruktoraufzug übergeben (so z. B. in Zeile 1179) und in der Folge nicht mehr geändert (die Instanzvariablen sind **private** deklariert und es gibt außer dem Konstruktor keine Methoden der Klassen, die schreibend darauf zugreifen, also die Instanzvariable auf der linken Seite einer Zuweisung stehen haben). Beim Stellen der Frage fügt **Tier** noch etwas Text zu dem Inhalt von **name** hinzu, so dass sich eine vollständige Frage ergibt; für Merkmale muss die Frage so eingegeben werden, wie sie hinterher gestellt wird.

Das Gros der Anwendungslogik steckt in der Klasse **Knoten**. Ihre Instanzen stellen die Knoten eines binären Baums, von denen der eine Nachfolger den Ja-, der andere den Nein-Zweig beinhaltet. Zudem muss jeder Knoten eine **Frage** haben; dass die entsprechende Instanzvariable **frage** heißt und vom Typ **Frage** ist, drückt aus, dass mit jedem Knoten entweder eine Tier- oder eine Merkmalsfrage verbunden ist. Dass nur die Blätter eines



Baums Tierfragen beinhalten dürfen, wird durch die Variablen Deklarationen nicht ausgedrückt; das steckt in der nachfolgenden Programmlogik, der Implementierung der Methode `fragen()`.

Die Methode `fragen()` enthält eine Unterscheidung von vier Fällen, die sich aus der Beantwortung der zu dem Knoten gehörenden Frage (ja oder nein) und dem Umstand, ob es sich um einen Blattknoten handelt (was man daran sehen kann, dass `ja` bzw. `nein` den Wert `null` haben) ergeben. Die Schachtelung der insgesamt drei If-else-Anweisungen ist Standard und hat mit Objektorientierung nichts zu tun. Objektorientiert ist dagegen die Fallunterscheidung, die sich hinter dem Aufruf `frage.stellen()` (Zeile 1201) verbirgt: Da `frage` eine Instanz der Klasse `Tier` oder `Merkmal` benennen kann, die beiden Klassen die Methode `stellen()` aber jede für sich implementieren, wird hier eine *Fallunterscheidung per dynamischem Binden* getroffen. Man beachte, dass die Unterscheidung, ob ein Knoten Blattknoten ist, ebenfalls per dynamisches Binden getroffen werden könnte; ihre Programmiererin hat sich aber im Rahmen ihrer kreativen Freiheit dagegen entschieden.

Vielleicht ist Ihnen aufgefallen, dass außer den Konstruktoren keine der Methoden einen Parameter hat. Es ist dies ein Zeichen für ein gelungenes objektorientiertes Design. Für Programmiererinnen, die aus der imperativen Programmierung kommen, ist dies gewöhnungsbedürftig — intuitiv denkt man zunächst, dass die Methoden stattdessen wohl auf globale Variablen zugreifen werden, weswegen man spontan die Nase rümpfen möchte. Das ist aber nicht der Fall: Alle Methoden greifen ausschließlich auf Instanzvariablen zu. Diese gibt es aber in der imperativen Programmierung nicht.

49 Lösungen zu den Selbsttestaufgaben

Selbsttestaufgabe 43.1 (Seite 232)

Es gibt keines. Man weiß nur, dass die Elemente von `liste Number` oder einem Subtyp davon als obere Schranke haben. Sobald es nicht mehr `Number` ist, können keine Instanzen vom Typ `Number` mehr zugewiesen werden, und je tiefer man den Subtyp wählt, desto weniger Typen können zugewiesen werden. Solange es keinen gemeinsamen tiefsten Subtyp aller Subtypen von `Number` gibt, können auch dessen Instanzen nicht als gültige Zuweisungen angenommen werden. Einzige Ausnahme: `null`.

Aber: Wenn `Number final` deklariert wäre (so dass es keine Subtypen von `Number` gäbe), dann könnte man auch `Number` als Elementtyp annehmen — dann wäre jedoch auch der ganze Umstand mit dem Typ-Wildcard nicht erforderlich.



Selbsttestaufgabe 43.2 (Seite 233)

Es ist weder möglich noch sinnvoll. Es wäre dann nämlich nicht klar, ob nun lesend oder schreibend auf die Elemente zugriffen werden darf.

Selbsttestaufgabe 43.3 (Seite 233)

Es ist!

Selbsttestaufgabe 43.4 (Seite 234)

```
1251 ArrayList<Object> o = new ArrayList<Object>();  
1252 ArrayList<String> s = new ArrayList<String>();
```

Und jetzt noch prüfen, ob `s.getClass() == o.getClass()` zu `true` auswertet, wenn man bloß wüsste, wie das in JAVA geht ... (s. Fußnote 3).

Selbsttestaufgabe 43.5 (Seite 236)

Sie gibt immer eine Liste zurück, deren Elemente den deklarierten Typ `Object` haben. Wenn man die Elemente anders verwenden will, muss man einen *Down cast* durchführen, der fehlschlagen (zu einem dynamischen Typfehler führen) kann.



Kurseinheit 5: Andere objektorientierte Programmiersprachen

A language that doesn't affect the way you think about programming,
is not worth knowing.

Alan Perlis

Während der obige Leitspruch aus akademischer Sicht sicher richtig ist, können sich Praktikerinnen (und solche, die es werden wollen) diese Einstellung nicht leisten. Stattdessen muss man die Sprachen kennen, für die es einen Markt gibt. Und so zeichnen sich die in dieser Kurseinheit behandelten Sprachen mit Ausnahme von EIFFEL weniger durch revolutionäre Konzepte oder neuartige Sichtweisen aus, sondern vielmehr dadurch, dass sie in der Praxis weite Verwendung finden. Gleichwohl mag die eine oder andere Eigenheit der einen oder anderen Sprache der Leserin Anlass geben, ihre bisherigen Denkweisen zu überprüfen.

Ziel dieser Kurseinheit ist es übrigens nicht, Sie zu Programmiererinnen in einer der (oder gar allen) in dieser Kurseinheit behandelten Sprachen C#, C++ und EIFFEL zu machen. Ziel ist vielmehr, Ihnen eine erste Übersicht und ein Verständnis dieser Sprachen zu vermitteln. Sie sollen einen Eindruck davon bekommen, auf was Sie sich einlassen, wenn Sie sich entscheiden, in einer der genannten Sprachen Software zu entwickeln. Dazu gehört nicht, jedes Konstrukt jeder dieser Sprachen zu kennen oder auch nur einmal gesehen zu haben, sondern vielmehr eine Vorstellung davon, was jeweils deren charakteristische Eigenschaften, was die Stärken und was die Schwächen jeder einzelnen Sprache sind. Je nachdem, wie sich die einzelnen Sprachen von bereits besprochenen unterscheiden, fällt die folgende Darstellung kürzer oder länger aus.

50 C#

C# ist MICROSOFTs Antwort auf JAVA. Oberflächlich betrachtet JAVA recht ähnlich, enthält C# doch einige zusätzliche Merkmale von C++, aber auch solche von MICROSOFTs Haussprache VISUAL BASIC. Dabei ist die Entwicklung von C# wohl weniger der Versuch, mit der MICROSOFT eigenen Marktmacht einen proprietären Standard durchzudrücken (was bei Programmiersprachen meines Wissens bislang auch noch nie gelungen wäre), sondern vielmehr dem Umstand geschuldet, dass JAVA keine volle Kontrolle über Rechner und Betriebssystem bietet und somit nicht für jede kommerzielle Softwareentwicklung geeignet ist. Zwar war auch



SUN bemüht, bekannte Schwächen zu beheben, aber man gibt sich dabei doch recht schwerfällig. Und so war, bei dem nicht zu übersehenden Erfolg JAVAS in der Softwareentwicklerinnengemeinde, die Abspaltung einer eigenen, JAVA-artigen, aber eben doch in wichtigen Punkten verschiedenen Sprache eigentlich nur folgerichtig.

50.1 Das Programmiermodell von C#

Das Programmiermodell von C# unterscheidet sich zunächst nicht wesentlich von dem JAVAS: Auch in C# ist der Code auf Klassen verteilt, die einzeln übersetzt werden können. Klassen werden in Dateien gespeichert, jedoch ist das Verhältnis von Klasse zu Datei lockerer als in JAVA (u. a. können Klassen anders heißen als ihre Dateien und sogar auf mehrere Dateien aufgeteilt werden). Allerdings sind der Bytecode und die dazu passende virtuelle Maschine nicht speziell für C# entworfen, sondern für alle sog. .NET-Sprachen. So heißt denn auch die Sprache des Bytecode *Common Intermediate Language (CIL)*; sie gilt als (gerade noch) menschenlesbar.



Anders als bei JAVA waren bei C# Flexibilität und Performanz von Anfang an kritische Gesichtspunkte des Sprachentwurfs. Für C# war daher von Anfang an und ausschließlich die sog. Just-in-time-(JIT-)Kompilierung vorgesehen, die den CIL-Code unmittelbar vor der Ausführung (und nur, wenn er überhaupt ausgeführt wird) in Maschinencode der Maschine, auf der er gerade läuft, übersetzt. Die Einheiten der JIT-Kompilierung gehen dabei hinunter bis zu einzelnen Methoden. Eine vollständige Kompilierung von CIL- in nativen Maschinencode vor der Ausführung ist ebenfalls möglich.

**keine Interpretation
von CIL-Code**

Eine andere Eigenschaft JAVAS, mit der die Programmiererinnen MICROSOFTS offenbar nicht unter allen Umständen leben konnten, ist die *Garbage collection*. In C# hat man daher die Möglichkeit, den Speicherplatz für Objekte, die mit `new` erzeugt wurden, selbst wieder freizugeben. Doch wehe der, die das vergisst: Speicherlecks sind die unmittelbare Folge. Noch schlimmer sind aber Speicherfreigaben von Objekten, auf die noch Referenzen existieren: Die zeigen dann ins Leere oder, wenn der Speicher wieder belegt wird, auf oder mitten hinein in ein anderes Objekt. Eine Katastrophe.

**programmgesteuerte
Speicherverwaltung**

Aber damit nicht genug: Das mit SMALLTALK und JAVA abgeschaffte Handtieren mit Pointern wurde in C# auch wiedereingeführt, wohl weil man in der systemnahen Programmierung (und bei Aufrufen in das hauseigene Betriebssystem) nicht darauf verzichten konnte. Allerdings sind beide Rückschritte — explizite Speicherverwaltung und das Handtieren mit Pointern — in sog. *unsichere Bereiche* verbannt. Dazu gibt es in C# einen Modifizierer `unsafe`, der solche Bereiche einleitet:

```
1253 public static unsafe void Swap(int* x, int* y) {  
1254     int temp = *x;  
1255     *x = *y;  
1256     *y = temp;  
1257 }
```

explizite Pointer



Dabei bedeutet der Stern hinter einem Typ, dass es sich um einen Zeiger-auf-Typ handelt; vor einer Variable bedeutet er, dass die Variable dereferenziert wird, also nicht auf den Pointer, sondern auf die Speicherstelle, auf die der Pointer zeigt, zugegriffen wird. Nebenbei bedeutet der Stern aber auch noch die Multiplikation und all das, wofür er sonst noch überladen wurde.

Neben Methoden können auch Klassen, Blöcke und Variablen unsicher sein.

50.2 Gemeinsamkeiten mit und kleinere Unterschiede zu JAVA

C# unterscheidet sich, was Objekte, Variablen und Ausdrücke, Anweisungen, Blöcke und Kontrollstrukturen angeht, nicht großartig von JAVA. Es ist in C# allerdings möglich, Operatoren (also z. B. +, -, == etc., aber nicht new, (), ||, &&, =) zu überladen. C# besitzt dafür das Schlüsselwort **operator**, das in einer Operatordefinition (die ansonsten so aussieht wie eine Methodendefinition) vorangestellt wird:

```
1258 static Matrix operator +(Matrix m1, Matrix m2) {...}
```

Außerdem ist es in C# Konvention, Methodennamen mit einem Großbuchstaben beginnen zu lassen, aber das ist wie gesagt nur Konvention. Wichtiger (und für viele Programmierprobleme von unschätzbarem Wert) ist da schon die Möglichkeit von C#, *Call by reference* nach dem Vorbild PASCALS (also ohne explizite Pointer; s. o.) zu erlauben und damit Funktionen wie das Vertauschen von Variableninhalten (die Methode swap) sicher zu programmieren:

| Call by reference |

```
1259 public static void Swap(ref int x, ref int y) {  
1260     int temp = x;  
1261     x = y;  
1262     y = temp;  
1263 }
```

Allerdings muss **ref** — anders als in PASCAL **var** — auch an der Aufrufstelle verwendet werden. Formale Parameter können auch mit **out** modifiziert werden (wobei für die Aufrufstelle dasselbe gilt wie für **ref**):

```
1264 public static void Aenderbar(out String s) {  
1265     s = "auch der tatsächliche Parameter ist geändert!"  
1266 }
```

Der Unterschied ist der, dass bei Verwendung von **ref** die Variable, die den tatsächlichen Parameter liefert, vor dem Aufruf initialisiert worden sein (einen Wert zugewiesen bekommen haben) muss, während dies bei **out** nicht der Fall ist. Dafür muss bei **out** der formale Parameter in der Methode einen Wert zugewiesen bekommen. Dass **ref** und **out** in C# anders als **var** in PASCAL an der Aufrufstelle wiederholt werden müssen, hat den Vorteil, dass die Programmiererin weiß, dass ihre die tatsächlichen Parameter liefernden Variablen nach dem Aufruf andere Werte haben können. Sie drücken also das Vorhandensein einer Zuweisung in beide Richtungen (hin und zurück) aus.



Selbsttestaufgabe 50.1

Überlegen Sie, welche Konsequenzen sich aus der Verwendung von `ref` bzw. `out` im Kontext des Subtyping ergeben.

Sowohl `ref` als auch `out` ermöglichen, dass eine Methode mehr als einen Rückgabewert hat. Da diese Möglichkeit in JAVA und SMALLTALK fehlt, findet man in diesen Sprachen häufig Klassen vor, die einzig dem Zweck dienen, mehrere Rückgabewerte in einem Objekt zu verpacken. Da sie an der Aufrufstelle aber wieder ausgepackt werden müssen, ist das eine ziemlich umständliche Lösung. Eine elegantere Alternative sind die Tupel EIFFELS (s. Abschnitt 52.7).

Nun verdient C# im Kontext von Methodenaufrufen nicht nur lobende Erwähnung. Die wohl bedeutendste Unterlassung ist, dass es in C# keine `Throws`-Klauseln in Methodendeklarationen gibt — die aus JAVA bekannte Unterscheidung von *Checked exceptions* und *Unchecked exceptions* (Abschnitt 47.2 in Kurseinheit 4) entfällt also und es gibt nur *Unchecked exceptions*. Das bedeutet, dass die Aufruferin einer Methode nicht gezwungen wird, darüber nachzudenken, was zu tun ist, wenn die Methode nicht korrekt ausgeführt werden kann; ja sie weiß nicht einmal bei Betrachten der Schnittstelle, dass die Methode auch abgebrochen werden kann. Das ist natürlich debattierbar, soll aber dem Umstand Rechnung tragen, dass bei einer stark geschichteten Architektur (beispielsweise beim Einsatz von Middleware) das Wissen um Exceptions auf der ganzen Wegstrecke von der Exception-Quelle bis zum Exception handler vorhanden sein muss, obwohl die mittleren Schichten naturgemäß an Art und Auftreten von Ausnahmen keinerlei Interesse haben. Das mit JAVA Version 1.4 eingeführte sog. *Exception chaining* erlaubt, eine Checked exception in einer Unchecked exception zu verpacken und später, z. B. nach Durchlaufen der Middleware, wieder auszupacken (erneut zu werfen). Das sog. *Exception tunneling* bietet ebenfalls Abhilfe.

keine Deklaration von Exceptions als Ergebnis eines Methodenaufrufs

Link



Ein weiterer, für die Programmierpraxis nicht weniger bedeutsamer Unterschied bei Methoden ergibt sich im Zusammenhang mit dem Überschreiben: Während in JAVA alle Methoden im Prinzip überschrieben werden können (es sei denn, ihre Definition trägt den Zusatz `final`), so dass der Compiler zunächst von einer dynamischen Bindung der Aufrufe ausgehen muss, sind in C#, der Tradition von C++ folgend, dynamisch zu bindende Methoden unbedingt als solche zu deklarieren, und zwar mit dem Schlüsselwort `virtual`. Entsprechend muss eine überschreibende Methode mit dem Schlüsselwort `override` deklariert werden. Soll hingegen eine Methode gleicher Signatur in einer Subklasse neu eingeführt (und nicht anstelle der, die sie überschreibt, dynamisch gebunden) werden, dann ist dies durch Verwendung des Schlüsselworts `new` bekanntzugeben. Anders als landläufig angenommen hat dies nicht nur Performanzgründe (es vermindert die Zahl der dynamischen Bindungen in einem Programm), sondern auch gewichtige programmiertechnische: Man markiert alle Stellen im Programm, an denen das sog. *Fragile-base-class-Problem* (Thema von Kapitel 55 in Kurseinheit 6) auftreten kann.

Deklaration von Überschreibung und dynamischem Binden

Link



Einige Sprachkonstrukte verwenden in C# andere Schlüsselwörter als JAVA, so **lock** anstatt **synchronized** sowie **foreach** anstatt **for** für die zweite Form von For-Schleifen (s. Kapitel 38 in Kurseinheit 4). Andere weichen in ihrer Bedeutung leicht von denen JAVAs ab: So sind auch Strings als Basis einer Switch-Anweisung zugelassen (in JAVA erst seit Version 7!) und jeder Zweig (case), der mindestens eine Anweisung enthält, muss mit einer expliziten Kontrollflussanweisung (**break**, **goto**, **return** oder **throw**) abgeschlossen werden. Außerdem hat C# eine Goto-Anweisung, mit der man jedoch nicht in Blöcke hinein springen kann. All dies hat allerdings nichts mit Objektorientierung zu tun.

50.3 Zusätzliche Ingredienzien von Klassendefinitionen in C#

Für die Programmierpraxis recht nützlich sind die in C# vorgesehenen zusätzlichen Elemente einer Klassendefinition. Es sind dies Properties, Indexer und Events.

50.3.1 Properties

Properties sind gewissermaßen die Umkehrung von Zugriffsmethoden (*Settern* und *Gettern*; s. Abschnitt 4.3.4 in Kurseinheit 1): Anstatt auf ein Feld eines Objektes über Methoden (lesend und schreibend) zuzugreifen, ruft man Methoden über das auf, was syntaktisch wie ein Feldzugriff aussieht. Dies erlaubt einer, (lesende und schreibende) Feldzugriffe mit Nebeneffekten zu versehen (wie z. B. einer dynamischen Typprüfung bei schreibendem Zugriff, wenn man kovariante Redefinition imitieren will). Konkret: Anstatt des (auch in JAVA üblichen)

```

1267 class Punkt {
1268     private float x, y;
1269
1270     void SetXY(float x, float y) {
1271         this.x = x;
1272         this.y = y;
1273     }
1274
1275     float GetX() {
1276         return x;
1277     }
1278
1279     float GetY() {
1280         return y;
1281     }
1282
1283     void SetRadiusWinkel(float r, float w) {
1284         x = Math.Cos(w) * r;
1285         y = Math.Sin(w) * r;
1286     }
1287
1288     float GetRadius() {
1289         return Math.Sqrt(x * x + y * y);
1290     }

```



```
1286     float GetWinkel() {
1287         return Math.Acos(x / r);
1288     }
1289 }
```

kann man in C# alternativ

```
1290 class Punkt {
1291     private float x, y;
1292
1293     float X {
1294         get {return x;}
1295         set {x = value;}
1296     }
1297
1298     float Y {
1299         get {return y;}
1300         set {y = value;}
1301
1302     float Radius {
1303         get {return Math.Sqrt(X * X + Y * Y);}
1304         set {
1305             float w = Winkel;
1306             x = Math.Cos(w) * value;
1307             y = Math.Sin(w) * value;
1308         }
1309     }
1310
1311     float Winkel {
1312         get {return Math.Acos(X / Radius);}
1313         set {
1314             float r = Radius;
1315             x = Math.Cos(value) * r;
1316             y = Math.Sin(value) * r;
1317         }
1318     }
1319 }
```

schreiben. Dabei sind `get` und `set` Schlüsselwörter von C# und `value` ist eine spezielle Variable (vergleichbar mit `this`), die den Eingabewert eines Setters hält. Um einem Punkt seine Koordinaten zuzuweisen bzw. darauf zuzugreifen, kann man dann die Properties `X`, `Y`, `Winkel` und `Radius` wie Felder verwenden:

```
1317 Punkt p = new Punkt();
1318 p.X = 1;
1319 p.Y = 1;
1320 if (p.Radius == Math.Sqrt(2) && p.Winkel == Math.PI/4) ...
```

Keine große Sache, aber es macht den Code auf Aufruferinnenseite knapper und besser lesbar. Den Getter oder den Setter kann man wahlweise auch weglassen; auf diese Weise lassen sich Felder mit Nur-lese- bzw. Nur-schreib-Zugriff simulieren.



50.3.2 Indexer

Indexer übertragen gewissermaßen das Konzept der indizierten Instanzvariablen von SMALLTALK auf C#: Jede Instanz einer Klasse, für die ein Indexer definiert ist, hat eine Menge von (scheinbar unbenannten) Instanzvariablen, die über einen Index zugegriffen werden können. Allerdings muss die indizierte Instanzvariable klassenintern durch eine normale, benannte Instanzvariable (Feld) repräsentiert werden; Indexer ähneln Properties insofern, als der Zugriff über einen Index mittels entsprechender Get- und Set-Abbildungen auf einen Zugriff auf eine benannte Instanzvariable übersetzt wird. Das folgende Beispiel illustriert den Vorgang:

```
1321 class HatScheinbarEineIndizierteInstanzvariable {
1322     private Object[] iivar;
1323
1324     HatScheinbarEineIndizierteInstanzvariable(int anzahl) {
1325         iivar = new Object[anzahl];
1326     }
1327
1328     Object this[int index] {
1329         get {return iivar[index];}
1330         set {iivar[index] = value;}
1331     }
1332 }
```

Dabei wird **this** als Schlüsselwort missbraucht, um anzudeuten, dass bei Zugriffen auf die indizierte Instanzvariable kein Name einer Instanzvariable (eines Feldes), sondern lediglich der Name des Objekts, zu dem sie gehört, steht:

```
1331 HatScheinbarEineIndizierteInstanzvariable einObject = new
          HatScheinbarEineIndizierteInstanzvariable(2);
1332 einObject[0] = einObject[1];
```

Nun darf der Indexer in C# überladen werden, so dass ein Objekt mehrere indizierte Instanzvariablen haben kann, wobei der Zugriff (aufgrund des fehlenden Namens) einzig über den Typ des Indexes differenziert erfolgen kann. Durch das Überladen ist es wiederum möglich, nicht eine, sondern mehrere indizierte Instanzvariablen zu simulieren, was jedoch der Beschränkung unterliegt, dass der Elementtyp (der Rückgabetyp beim Überladen) gleich bleiben muss. Und schließlich muss ein Indexer auf keine interne (benannte) Instanzvariable zugreifen — alle Inhalte können, genau wie bei Properties, auch berechnet werden.

50.3.3 Ereignisse (Events)

Viele Applikationen, insbesondere solche mit GUI, benötigen neben der direkten Kommunikation zwischen Objekten, die sich kennen (die ja durch Nachrichtenaustausch bzw., je nach Diktion, durch Methodenaufrufe bewerkstelligt wird), auch eine Kommunikation mit unbekannten. Die Problematik hatten wir im Kontext von SMALLTALK bereits besprochen (Abschnitt 14.3 in Kurseinheit 1).



Nun kommt dieses Problem so häufig vor, dass man sich bei MICROSOFT dafür entschieden hat, es zumindest teilweise von der Ebene der Programmierung (wo es in Form eines sog. *Patterns* abgehandelt wird; mehr dazu in Kurs 01853) auf die Ebene der Programmiersprache zu heben (in SMALLTALK, wo diese Unterscheidung nicht so ausgeprägt ist, war das Problem ja mittels einer Implementierung der benötigten Mechanismen in der Klasse **Object**, von der alle anderen erben, gelöst worden). Es wurde zu diesem Zweck das Konstrukt des **Events** (Ereignisses) eingeführt, über das sog. *Event handler* aktiviert werden können. Dabei sind die Event handler in Abschnitt 50.4.1 skizzierte sog. *Delegates*. Leider ist die mit Deklaration und Registrierung von Event handlern sowie der Verbreitung von Ereignissen verbundene Syntax von C# nach Ansicht des Autors dieses Textes komplett unleserlich geraten, so dass hier auf eine weitergehende Befassung mit dem Thema verzichtet wird.

50.4 Das Typsystem von C#

Auch wenn es in großen Teilen recht ähnlich ausfällt, so weicht das Typsystem von C# von dem JAVAs gleich in mehreren wesentlichen Punkten ab:

- der Art der Unterscheidung von Wert- und Referenztypen,
- den angebotenen Typkonstruktoren für Wert- und Referenztypen und
- dem Umgang mit Interfaces als Typen.

Darüber hinaus hat C# noch eine ganze Reihe weiterer Verbesserungen, die man mit dem Typsystem in Verbindung bringen kann; auf sie wird hier aber nur am Rande eingegangen.

50.4.1 Die Typhierarchie von C#

In C# sind genau wie in JAVA alle Variablen typisiert. Anders als in JAVA wird dabei jedoch zunächst nicht zwischen Wert- (primitiven) und Referenztypen unterschieden: Alle Typen, auch die primitiven, gelten als von **Object** (genauer: **System.Object**) abgeleitet. Da lohnt es sich, auf die Typhierarchie etwas genauer einzugehen.

Genaugenommen ist die Typhierarchie von C# gar nicht die Typhierarchie von C#, sondern die von .NET: Sie ist nämlich für alle .NET-Sprachen dieselbe. Das liegt daran, dass .NET für alle seine Sprachen ein gemeinsames Typsystem vor sieht, nämlich das *Common Type System* (CTS). Das CTS sorgt dafür, dass Typen, die in einer Sprache definiert wurden, auch in einer anderen Sprache verwendet werden können, und zwar ganz so, als seien sie in der anderen Sprache selbst definiert worden. Wie man sich leicht vorstellen kann, sind dafür einige Konventionen notwendig.

Common Type
System

Das erste Merkmal des CTS ist, dass *alle Typen* in einer Hierarchie untergebracht sind. Die aus JAVA bekannte Ausgrenzung der primitiven Typen gibt es also nicht. Tatsächlich sind die primitiven Typen als eine von mehreren Arten von

Einordnung von
Werttypen



Werttypen in der Hierarchie angesiedelt. Eine weitere wichtige Form von Werttypen sind die (aus PASCAL bekannten und inzwischen auch in JAVA, dort aber als Referenztypen angekommen) Aufzählungstypen, deren Elemente (Werte) von der Programmiererin selbst angegeben werden können (im Gegensatz zu denen der primitiven Typen, deren Werte mit der Sprachdefinition vorgegeben sind):

```
1333 enum Colour { Red, Blue, Green }
```

Aus Werttypen können, genau wie in PASCAL oder C, mittels des Typkonstruktors **struct** (dem C-Äquivalent von PASCALS **record**) neue Werttypen erzeugt werden, die sogar Methoden und Konstruktoren haben können, die aber keine Klassen sind (und insbesondere keine *Typerweiterung* und somit auch keine *Vererbung* erlauben):

Werttypkonstruktion mit struct

```
1334 struct Point {  
1335     int x, y;  
  
1336     void move(int x, int y) {...}  
1337 }
```

Man erspart sich durch das Weglassen der Typerweiterung die Projektion bei Zuweisungen, also das Fallenlassen von Feldern, die ein erweiterter Typ hinzugefügt hat und für die im für die Zielvariable reservierten Speicher kein Platz ist (vgl. Kapitel 23 in Kurseinheit 2, insbesondere Fußnote 50). Der Typkonstruktor „Array von“ ([]) führt in C# jedoch, genau wie in JAVA, zu einem Referenztypen.

Bei den Referenztypen wird dann das zweite wichtige Unterscheidungsmerkmal sichtbar: Neben Klassen, Interfaces und Arrays gibt es auch noch sog. **Delegates**, das sind im wesentlichen (Zeiger auf) an ein Objekt gebundene, einzelne Methoden. Delegates ersetzen die aus anderen Sprachen bekannten Funktionspointer; sie konnten in JAVA bis Version 8 nur recht umständlich über Interfaces und anonyme innere Klassen emuliert werden, an deren Stelle heute freilich die Lambda-Ausdrücke JAVAs treten können (s. Kapitel 37 in Kurseinheit 4). Delegates sind für verschiedene Problemstellungen (z. B. Listener-Mechanismen) sehr nützlich.

Delegates als weitere Art von Referenztypen

Zuletzt gibt es in C# auch noch sog. Attribut-Typen (**Attributes**); sie entsprechen im wesentlichen den *Annotationen*, die es seit der Version 5 auch in JAVA gibt. Annotationen haben aber mit objektorientierter Programmierung nicht unmittelbar etwas zu tun und sind daher nicht Gegenstand dieses Kurses (s. a. Abschnitt 47.4 in Kurseinheit 4). Es sei nur soviel erwähnt, dass in C# ein Attribut namens **Serializable** das gleichnamige *Marker-Interface* JAVAs ersetzt (vgl. Kapitel 45 in Kurseinheit 4).

Attributes

Es ergibt sich die folgende grobe Einteilung der Typen von C#:

Einteilung der Typen von C#



Werttypen	Referenztypen	Attribute
• primitive Typen	• Klassen	
• Aufzählungstypen	• Interfaces	
• Strukturtypen (Records)	• Arrays	
	• Delegates	

Wohlgemerkt, dies ist keine Klassenhierarchie, sondern lediglich eine Einteilung der verschiedenen Arten von Typen in C#. Die Klassenhierarchie ist wesentlich komplexer und vereinheitlicht zudem, wie ja bereits gesagt, das Typsystem von C# (wie auch das CTS), indem alle Typen von `System.Object` ableiten. Die Klassenhierarchie im Namespace `System` fängt dann auch ungefähr so an (Subklassen sind eingerückt):

```

Object
  Array
  Attribute
  Delegate
    MulticastDelegate
    AsyncCallback
    EventHandler
  Exception
  ValueType
    Boolean
    Enum
      DayOfWeek
    Guid
    TimeSpan
    TypedReference
    Void
  
```

Klassenhierarchie

Wie Ihnen sicher aufgefallen ist, sind einige der Arten von Typen aus der obigen Liste jeweils durch eine spezielle Klasse (`Array`, `Delegate`, `Attribute`) vertreten. Man kann dies als Hinweis darauf verstehen, dass tatsächlich alle Arten von Typen integriert sind und es keine grundsätzlichen Barrieren zwischen ihnen gibt. In C# haben übrigens alle Klassen außer `System.Object` ganz wie in JAVA genau eine Superklasse; sie können aber (auch wie in JAVA) beliebig viele Interfaces implementieren. Die Tatsache, dass Werttypen als Subtypen eines Referenztypen (nämlich `Object`) deklariert sind, verrät außerdem, dass C# über *Auto boxing* und *unboxing* verfügt.

50.4.2 Interfacetypen in C#

Zwar hat C# das Interface-als-Typ-Konzept von JAVA übernommen, doch hat man hier seine Rolle deutlich gestärkt. So ist es in C# möglich, dass von verschiedenen Interfaces „geerbte“, gleiche Methodendeklarationen in einer Klasse getrennt voneinander implementiert werden können. Dies geschieht mit sog. **expliziten Interfaceimplementierungen**, wie im



folgenden Beispiel dargestellt (man beachte, dass in C# der Doppelpunkt die Schlüsselwörter `extends` und `implements` ersetzt⁷⁵):

```
1338 interface Angestellte {  
1339     Telefonnummer gibTelefonnummer();  
1340 }  
  
1341 interface Privatperson {  
1342     Telefonnummer gibTelefonnummer();  
1343 }  
  
1344 class Person : Angestellte, Privatperson {  
1345     private Telefonnummer privat, büro;  
  
1346     Telefonnummer Angestellte .gibTelefonnummer() {  
1347         return büro;  
1348     }  
  
1349     Telefonnummer Privatperson.gibTelefonnummer() {  
1350         return privat;  
1351     }  
1352 }
```

Der Nutzen der expliziten Interfaceimplementierung in den Zeilen 1346–1351 ist der, dass ein Objekt der Klasse `Person` auf die gleiche Nachricht `gibTelefonnummer()` unterschiedlich reagiert, und zwar abhängig davon, über welches Interface es angesprochen wird. Der Typ der Variable (oder des Ausdrucks), die als Empfänger fungiert, gibt also gewissermaßen die Rolle (hier: `Angestellte` bzw. `Privatperson`) vor, in der das Objekt angesprochen wird. Es ist weder vorgesehen noch möglich, dass man an der Aufrufstelle etwas wie `x.Angestellte.gibTelefonnummer()` schreibt (wobei `x` das Objekt bezeichnet); vielmehr steht dort einfach nur `x.gibTelefonnummer()`. Falls übrigens die Methodendeklaration `Telefonnummer gibTelefonnummer()` von einem gemeinsamen Superinterface von `Angestellte` und `Privatperson`, z. B. `Erreichbar`, geerbt würde, wäre das Programm ungültig; es muss nämlich immer der tatsächlich deklarierende Typ als Qualifizierer angegeben werden.

Wenn im obigen Beispiel für die beiden expliziten Interfaceimplementierungen von `gibTelefonnummer()` nicht wie in JAVA verlangt der Zugriffsmodifikator `public` angegeben wurde, dann geschah das nicht ohne Grund: Es ist in C# nämlich möglich, Methoden nicht `public` zu deklarieren und trotzdem, per Interfaceimplementierung, von außen zugreifbar zu haben. Allerdings ist dies an die explizite Interfaceimplementierung gebunden. Die sog. **implizite Interfaceimplementierung**, also

explizite Interface-implementierung

⁷⁵ Der (berechtigten) Kritik, dass dies eine wichtige Unterscheidung verwischt und man so bei Be trachtung einer Klasse manchmal nicht sagen kann, welcher der aufgelisteten Supertypen Interface und welcher Superklasse ist, wird eine Namenskonvention entgegen gehalten: Im Common Type System von .NET sollten alle Interfacenamen mit einem „I“ beginnen. Das steht in der Tradition der bei MICROSOFT weithin gebrauchten und nach dem früheren Mitarbeiter CHARLES SIMONYI so genannten ungarischen Notation.



die, die Sie von JAVA her kennen, ist in C# natürlich auch vorgesehen; dort müssen Methoden jedoch immer (wie auch in JAVA) **public** deklariert werden.

Explizite Interfaceimplementierungen können nicht überschrieben werden (dürfen also auch nicht **virtual** deklariert sein). Eine Klasse, die von einer mit expliziten Interfaceimplementierungen erbt, erbt diese also, ohne dass sie diese überschreiben könnte — es sei denn, sie „reimplementiert“ das Interface, d. h., sie deklariert selbst, es zu implementieren. Man beachte aber, dass diese erneute explizite Implementierung keine Überschreibung darstellt: insbesondere findet bei Aufruf der explizit implementierten Methode kein dynamisches Binden statt.

**mangelnde
Überschreibbarkeit
expliziter Interface-
implementierungen**

50.4.3 Generizität in C#

Genau wie in JAVA entspricht in C# zunächst jeder Klasse und jedem Interface ein Typ. Mit der Version 2.0 ist C# aber ebenfalls generisch geworden. Genauer gesagt erlaubt C# sowohl beschränkten als auch unbeschränkten parametrischen Polymorphismus, sowohl von Klassen als auch von Methoden. Die Syntax für beschränkte Typparameter sieht so aus:

1353 **class** SortedList<E> **where** E : IComparable ...

die für unbeschränkte unterscheidet sich nicht von der JAVAs. Parametrisch definierte Klassen und Interfaces spezifizieren jeweils eine (potentiell unendliche) Menge von Typen, die durch Einsetzen konkreter (tatsächlicher) Typparameter in die Typvariablen entstehen. C# erlaubt zudem (genau wie JAVA; s. Abschnitt 43.5 in Kurseinheit 4), den tatsächlichen Typparameter bei parametrischen Methodenaufrufen wegzulassen, wenn ihn der Compiler aus den Typen der Argumente erschließen kann (*Typinferenz*).

Auch wenn sich die Generics von C# auf den ersten Blick nicht groß von denen JAVAs zu unterscheiden scheinen, so verbirgt sich hinter der Oberfläche doch ein anderer Mechanismus. Während JAVA die Typparameter grundsätzlich immer wegkompliert (um Abwärtskompatibilität zu erreichen; die sog. *Type erasure*), instanziert C# im Fall von Werttyp-Parametern (also **int**, **float** etc., aber auch die per **struct** definierten Typen) jede generische Klasse für jeden verwendeten Typ einmal, erzeugt also alternative Implementierungen (sog. *Typexpansion*). Dies hat den Vorteil, dass diese Implementierungen ohne *Boxing/Unboxing* auskommen und vom JIT-Compiler per Berücksichtigung der Typparameter optimiert werden können. Für Referenztypen wird der Code jedoch (wie in JAVA) nur einmal erzeugt. Gleichwohl bleibt die generische Typinformation in C# auch zur Laufzeit erhalten und kann per *Reflection* abgefragt werden.

**Unterschiede der
Generizität zwischen
C# und JAVA**

Genau wie in JAVA gibt es in C# beim Subtyping von Containern (wie z. B. Collections) ein Varianzproblem: **Collection<A>** und **Collection** sind auch dann nicht zuweisungskompatibel, wen A ein Subtyp von B ist (vgl. Abschnitt 43.2 in Kurseinheit 4). Um dennoch Zuweisungskompatibilität herzustellen, sieht C# keine spezielle Annotation der *Benutzung*



eines Typs wie in JAVA vor, sondern eine Annotation der *Definition* des Typs⁷⁶: Dem `ko-` bzw. `kontravarianten` Typparameter wird dazu das Schlüsselwort `out` bzw. `in` vorangestellt. Die Beschränkungen (nur lesen bzw. nur schreiben) sind dann bei allen Verwendungen des Typs die gleichen. Übrigens: Für Arrays in C# gilt dasselbe wie in JAVA: Sie sind kovariant, das Schreiben in ein Array kann aber zu einem Laufzeittypfehler führen, der in C# „ArrayTypeMismatchException“ heißt.

50.4.4 Die dynamische Komponente

C# soll genau wie JAVA und anders als C++ eine typsichere Sprache sein, also eine strikte Typprüfung durchführen. Da aber (ebenfalls genau wie in JAVA) nicht alles zur Übersetzungszeit geschehen kann, hat auch das Typsystem von C# eine Laufzeitkomponente.

Um einen Ausdruck einer Typumwandlung zu unterziehen, bietet C# genau wie JAVA und C++ Casts an. Auch die Syntax unterscheidet sich nicht:

| **Typumwandlung** |

1354 `(T) a`

bewirkt, dass der Ausdruck `a` den Typ `T` aufgedrückt bekommt. Ist dies nicht möglich, weil der tatsächliche Typ des Objektes, auf das `a` verweist, kein Subtyp von `T` ist oder weil keine entsprechende Typumwandlung definiert ist (einschl. Boxing/Unboxing), wird dies mit einem Laufzeitfehler quittiert. Casts sind also typsicher (in dem Sinne, dass keiner Variable ein Wert zugewiesen wird, den sie nicht haben darf), aber nicht sicher (sie können zu Ausnahmesituationen und, im Falle einer Nichtbehandlung, zu Programmabbrüchen führen).

Um Casts sicher zu machen, bietet C# den Operator `is` an. Er entspricht im wesentlichen dem `instanceof` von JAVA und gibt für einen Ausdruck der Art

| **sichere** | **Typumwandlung** |

1355 `a is T`

wobei `a` für einen beliebigen Ausdruck und `T` für einen Typ steht, zurück, ob das Ergebnis der Auswertung von `a` mit einer Variable vom Typ `T` zuweisungskompatibel ist. Dabei wird sowohl das Subtyping als auch das implizite (Auto-)Boxing berücksichtigt. Programmfragmente der Art

```
1356 T x;  
1357 if (a is T) {  
1358     x = (T) a;  
1359     ...
```

sind also sicher. Parallel zum Operator `is` gibt es noch einen weiteren `as`, der die typsichere Zuweisung erlaubt:

⁷⁶ weswegen man die Form der Varianz auch *declaration-site variance* nennt.



wobei a wieder für einen beliebigen Ausdruck und T für einen Typ steht, verursacht nie einen Laufzeitfehler, weil bei mangelnder Zuweisungskompatibilität einfach **null** (das mit allen Variablen zuweisungskompatibel ist) eingesetzt wird. Das obige Programmfragment (Zeilen 1356–1359) ist demnach mit dem folgenden fast äquivalent:

```
1361 T x;
1362 x = a as T;
1363 if (x != null) {
1364     ...
```

50.5 Ausblick

C# ist nicht nur Nachahmer JAVAs, sondern war auch Pionier für Sprach-**erweiterungen**, die es inzwischen auch in JAVA gibt: So gab es in C# von Anfang an sog. Attributes, die in JAVA in der Version 5 als Annotationen (was in jedem Fall der bessere Name ist) Einzug gehalten haben. C# hatte auch schon ab der Version 3.0 *Lambda-Ausdrücke* (ein Element *funktionaler Programmiersprachen*, in SMALLTALK — in Form von *Blöcken* — schon immer vorhanden und ein Grundelement der Sprache); JAVA ist erst mit Version 8 nachgezogen. Die Gefahr ist jedoch, dass sich die beiden Sprachen gegenseitig totrüsten — irgendwann wird jemand auf die Idee kommen, aus den besten Eigenschaften beider eine neue Sprache zu destillieren (KOTLIN mag hier ein erstes Beispiel sein). Vielleicht wird bei der Gelegenheit ja auch endlich mit dem C-Erbe aufgeräumt und die fürchterliche Syntax entsorgt.

C# als Vorreiter

51 C++

With syntax so chaotic that even compilers have to guess at it, C++ code had better be reusable, because no one will ever want to reverse-engineer it. The programming language's "feature" — being a superset of C — is a fundamental bug. With numerous large projects being written in already obsolete dialects, C++ is arguably an "instant legacy" language.

James Hopper



Über C++ ist viel geschrieben worden, nicht alles davon positiv im Tenor. Dabei wird jedoch häufig vergessen, dass eine der harten Nebenbedingungen des Entwurfs von C++, die vollständige Rückwärtskompatibilität mit C, ein derart schweres Handicap darstellt, dass nahezu jede Kritik an C++ als unfair gelten muss.⁷⁷ Natürlich kann man in C++ komplett unlesbare Programme schreiben, aber das gilt schon deswegen, weil man in C komplett unlesbare Programme schreiben kann. Man kann aber auch C++ mit einer neuen Syntax versehen (mittels seines Präprozessors, der übrigens Turing-äquivalent ist, also die Ausdrucksstärke einer vollwertigen Programmiersprache besitzt) und dann darin komplett lesbare Programme schreiben. Wenn man denn will.



51.1 Das Programmiermodell von C++

Das Programmiermodell von C++ ist ein klassisches: Programme werden als Menge von Quellcode-Dateien geschrieben, die in auf einer Zielmaschine direkt ausführbaren Maschinencode übersetzt werden. Getrennte Übersetzung von Programmteilen ist dank sog. Header files, die die Schnittstellen der Teile enthalten, möglich. Getrennt übersetzte Programmteile müssen vor der Ausführung gebunden werden; dynamisches (Nach-)Laden von Funktionen ist möglich, muss aber explizit (programmgesteuert) erfolgen.

C++ ist als objektorientierter Nachfolger von C konzipiert und soll so einen stufenlosen Übergang von der prozeduralen zur objektorientierten Programmierung ermöglichen. Dies wird insbesondere für die Migration von Altsystemen hin zur Objektorientierung als nützlich erachtet. Entsprechend zielt C++ auf die gleiche Klasse von Anwendungen wie C ab: maschinennahe Programmierung wie die von Betriebs- oder eingebetteten Systemen. Extreme Speicher- und Recheneffizienz sind dabei häufig oberste Kriterien.

51.2 Klassen

C++ ist insofern objektorientiert, als es neben den aus C übernommen Strukturen (structs) auch Klassen anbietet. Diese beinhalten, genau wie in SMALLTALK und JAVA, neben Feldern (Instanzvariablen) auch Methoden. Klassenfelder und -methoden werden (wie in JAVA; s. Abschnitt 36.1 in Kurseinheit 4) mit dem Schlüsselwort **static** in einer Klasse eingeführt. *Metaklassen* gibt es in C++ nicht; gleichwohl kann der Name einer Klasse als Wert verwendet werden.

Dass man in C++ wie in JAVA das Schlüsselwort **class** verwendet, heißt nicht automatisch, dass man damit Klassen im Sinne JAVAs oder SMALL-

Klassen haben Wertsemantik

⁷⁷ Sie kennen vielleicht auch den Witz, nach dem C von Amerikanern mit dem Ziel entwickelt wurde, es den Russen zuzuspielen, um deren Informatik um Jahrzehnte zurückzuwerfen, und dass die Entwicklung der Sprache erst als abgeschlossen erachtet wurde, nachdem sich `for(;P("\n"), R--;P("|")) for(e=C;e--;P("_"+(*u++/8)%2)) P("| "+(*u/4)%2);` übersetzen ließ.



TALKs definiert. Insbesondere haben Variablen mit einer Klasse als Typ keine *Referenz*, sondern *Wertsemantik*. Entsprechend müssen die Werte solcher Variablen, die „Objekte“, nicht erst mit **new** angelegt werden — der für ein „Objekt“ benötigte Speicherplatz wird, genau wie bei den Records PASCALS oder bei den Structs von C, bei der Deklaration reserviert. Dabei steht „Objekt“ hier deswegen in Anführungsstrichen, weil diese „Objekte“ eigentlich keine Objekte sind, sondern Werte; insbesondere haben sie keine *Identität* und bei Zuweisungen an andere Variablen werden Kopien angefertigt. Aliase gibt es entsprechend zunächst auch keine.

Um in C++ Objekte mit Identität zu erzeugen, muss man Variablen vom Typ eines Zeigers auf eine Klasse anlegen und dann eine Klasse mit dem New-Operator instanziieren. Syntaktisch sieht das, wenn man einen parameterlosen Konstruktor für die Klasse A als gegeben voraussetzt, so aus:

Hantieren mit Pointern

```
1365 A* a = new A();
```

Sieht man einmal von der expliziten Festlegung, dass es sich bei der Variable **a** um eine Pointervariable handelt, ab, dann gleicht diese Anweisung einer äquivalenten in JAVA oder C#.

Zur Dereferenzierung einer solchen Pointer-(Objekt-)Variable bei gleichzeitigem Zugriff auf ein Element (Feld oder Methode) der Instanz schreibt man in C++

```
1366 a->x
```

bzw.

```
1367 a->f(x)
```

wobei **x** ein Feld und **f(.)** eine Methode der Klasse A sein soll. Dies ist äquivalent zu **(*a).x** bzw. **(*a).f(x)**. Besondere Obacht ist bei Zuweisungen geboten, da man sich hier genau überlegen muss, ob man Pointer oder die Werte, auf die die Pointer zeigen, zuweisen will.

Komplikation bei impliziten Zuweisungen

Besonders verwirrend ist die Situation bei den *impliziten Zuweisungen*, die im Rahmen von Methodenaufrufen stattfinden. C++ macht zunächst ein *Call by value*, das heißt, es wird eine Kopie des Inhalts des tatsächlichen Parameters dem formalen Parameter zugewiesen. Im Falle von Pointer-Variablen wie dem obigen **a** bedeutet das aber, dass nicht das Objekt, sondern nur der Zeiger auf das Objekt übergeben wird. Es entspricht dies genau dem Verhalten von JAVA und SMALLTALK, wobei allerdings bei beiden nirgends explizit angegeben wird, dass es sich um eine Pointervariable handelt — es ist einfach immer so.



Will man nun davon abweichend ein *Call by reference* haben, dann gibt es zum einen die Möglichkeit, an der Aufrufstelle den Zeigeroperator & zu verwenden, der anstelle einer Kopie des Inhalts der Variable einen Zeiger auf die Speicherstelle der Variable erzeugt und diesen übergibt:

```
1368 A* a, b;  
1369 swap(&a,&b);
```

Dafür müssen dann aber die formalen Parameter so deklariert werden, dass sie Zeiger auf Zeiger aufnehmen können, also etwa wie in

```
1370 void swap(A** a,A** b) {  
1371     A* tmp=*a;  
1372     *a=*b;  
1373     *b=tmp;  
1374 }
```

Alternativ gibt es in C++ aber die Möglichkeit, wie in PASCAL zu verfahren und einfach

```
1375 void swap(A* &a, A* &b) {  
1376     A* tmp=a;  
1377     a=b;  
1378     b=tmp;  
1379 }
```

zu schreiben, wobei dann die Aufrufstelle unverändert bleiben kann (also ohne & auskommt). Vgl. dazu aber die Bemerkungen in Abschnitt 50.2 zur Praxis in C#.

51.3 Friends

Ein interessantes Konzept von C++, das einen direkten Bezug zur objektorientierten Programmierung hat, ist das Friends-Konzept. In der Praxis kommt es häufig vor, dass ein bestimmtes Teilproblem nicht von einer Klasse allein, sondern nur durch das Zusammenspiel mehrerer Klassen gelöst werden kann. Während diese Klassen untereinander eng kooperieren müssen und deswegen (relativ) intime Kenntnis voneinander benötigen (will sagen, auf Elemente zugreifen können müssen, die anderen Klassen verborgen bleiben sollten), gilt das für andere Klassen nicht unbedingt. Die Schnittstelle solcher kooperierenden Klassen sollte also nicht absolut, sondern relativ zu anderen Klassen definierbar sein.

In JAVA hatte man dazu bis zur Version 8 nur die Möglichkeit, die besagten Klassen in ein Paket zu verfrachten und die fraglichen Elemente mit paketweitem Zugriff (also ohne Zugriffsmodifizierer) zu deklarieren (Abschnitt 39.1 in Kurseinheit 4). Das hat jedoch den Nachteil, dass alle Klassen desselben Pakets dieselbe Schnittstelle jeder einzelnen enthaltenen Klasse haben; wenn es eine Klasse gibt, die eine ansonsten unsichtbare Eigenschaft x einer Klasse A sehen und eine andere, die eine Eigenschaft y derselben Klasse sehen soll, dann gibt es keine Aufteilung der Klassen auf Pakete, die genau dieses gestattet. Was man stattdessen gern hätte, ist ein dedizierter Export von Elementen einer Klasse, also ein

| dedizierter Export |

zur Verfügung Stellen von Elementen an genau benannte Klassen. Dies bietet, in etwas größerer Form, das Friends-Konzept von C++: Die Definition

```
1380 class A {  
1381     friend class B;  
1382     friend class C;  
1383     ...  
1384 }
```

bewirkt, dass (Instanzen von) B und (von) C auf alle **private** deklarierten Elemente von (Instanzen von) A zugreifen können (jedoch weder umgekehrt von A auf B und C noch B und C gegenseitig). Ein spezifischer, dedizierter Export von einzelnen Membern an bestimmte Klassen ist in C++ nicht möglich; diesen gibt es dafür in EIFFEL (s. Abschnitt 52.2). Mit JAVAs Modulen und dem dedizierten (*qualifiziert* genannten) Export wird das Friend-Konzept auf Ebene der Pakete zumindest angenähert.

51.4 Mehrfachvererbung

Getreu seinem Motto, alle Freiheit in die Hand der Programmiererin zu legen und ihr nicht mit einer gouvernantenhaften Du-kannst-das-bestimmt-nicht-Attitüde zu begegnen, bietet C++ (im Gegensatz zu SMALLTALK, JAVA und C#) uneingeschränkte **Mehrfachvererbung**. Das kann aus verschiedenen Gründen sinnvoll erscheinen:

Eine Klasse, die von mehreren, vollständig *abstrakten* (also mit keinerlei Implementierung versehenen) *Klassen* erbt, implementiert damit faktisch mehrere Interfaces.

Der einzige Nachteil ist, dass die Programmiererin nicht in Mitteln der Sprache ausdrücken kann, ob eine abstrakte Klasse die Funktion eines Interfaces oder die einer Generalisierung (von der man Implementierung erben kann; vgl. Abschnitt 9.1) haben soll — dazu sind dann schon Namenskonventionen notwendig.

- Nicht selten ergibt sich aus der Aufgabenstellung, dass eine Klasse Eigenschaften von mehreren anderen gebrauchen könnte. In Sprachen mit Einfachvererbung muss man sich dann für eine Klasse als Superklasse entscheiden und den Beitrag der anderen Klassen wiederholen, also erneut implementieren oder per *Delegation* bzw. *Forwarding* einbinden.⁷⁸ Mehrfachvererbung erlaubt im Gegensatz dazu, sich alles zusammenzuerben, was man benötigt.

Da das Erben jedoch nicht selektiv (in dem Sinne, dass man sich aussuchen könnte, was man von einer Klasse erbt) erfolgt und das Löschen von geerbten Membern in C++ nicht möglich ist, fühlt man sich häufig bemüßigt, die Klassen, von denen man erbt, in viele kleine

⁷⁸ Bei der Delegation, oder genauer beim Forwarding, ordnet man der Instanz der Hauptklasse Instanzen der anderen Klassen bei und delegiert die Funktionen, die die Hauptklasse nicht erben kann, an die beigeordneten Instanzen. Das ist dann jedoch jedes mal auszuprogrammieren und daher ziemlich lästig.



aufzusplitten und nur die zu beerben, deren Eigenschaften man braucht, um sich von unnötigem Ballast freizuhalten.

Mehrfachvererbung ist etwas, das sich Programmiererinnen gerne wünschen. Sie bringt jedoch einige praktische Probleme mit sich, unter anderem die Frage, was zu tun ist, wenn eine Klasse von mehreren anderen Klassen verschiedene Definitionen desselben Elements (Feld oder Methode) erbt. Da die Klasse sich dann für eine der beiden Definitionen entscheiden muss, geht die der anderen verloren. Dies kann, insbesondere im Zusammenhang mit *dynamischem Binden* und *offener Rekursion*, zu unerwartetem Verhalten führen. Darüber hinaus führt die Mehrfachvererbung noch zu zahlreichen weiteren Problemen, die hier nicht weiter ausgeführt werden sollen.

Problem der Mehrfachvererbung

51.5 Das Typsystem von C++

Das Typsystem von C++ stellt den Versuch dar, objektorientierte Programmierung mit starker Typsicherheit unter Beibehaltung der vollen Freiheit der Programmiererin mit möglichst wenig Laufzeit-Overhead zu erzielen. Dazu gibt es eigentlich nur einen Kommentar:

More computing sins are committed in the name of efficiency (without necessarily achieving it) than for any other single reason — including blind stupidity.

W.A. Wulf

51.5.1 Statische Komponente

Wie bereits eingangs erwähnt, wurde bei der Definition von C++ als objektorientierte Erweiterung der Sprache C stark auf Rückwärtskompatibilität geachtet. Und so findet sich auch das Typsystem Cs vollständig in C++ wieder. Es gibt also primitive Datentypen wie `int`, `bool` etc. und auch die Typkonstruktoren `struct` (entsprechend dem `record` PASCALS) und `union` (entsprechend dem varianten Record). Alle diese Typen sind, genau wie die durch Klassen definierten, Typen mit Wertsemantik. Es lassen sich aber auch, genau wie in C, Zeigertypen darauf definieren.

Wie in Abschnitt 51.4 diskutiert, erlaubt C++ anders als alle zuvor diskutierten Sprachen Mehrfachvererbung. Es kann also eine Klasse von mehreren anderen abgeleitet werden. Die Syntax von C++ sieht (analog zur mehrfachen Interface-Implementierung bei C#) dazu vor, die Namen der Basisklassen, von denen abgeleitet wird, durch einen Doppelpunkt getrennt hinter dem Namen der zu definierenden Klasse aufzulisten, wie in

Klassenhierarchie



Die Mehrfachvererbung wirkt sich natürlich auch auf das Typsystem aus: Ein von einer Klasse abgeleiteter Typ kann beliebig viele direkte Supertypen haben (nämlich einen pro direkte Superklasse). C++ kennt dafür keine Interfaces wie JAVA oder C#; sie müssen durch rein *abstrakte Klassen* emuliert werden. Ein Problem ergibt sich dann, wenn verschiedene Supertypen eines Typs dieselbe Eigenschaft anders spezifizieren — der Subtyp hat dann einen Konflikt, da er nicht den Spezifikationen beider Supertypen dienen kann.

Obwohl die Zuweisungskompatibilität in C++ wie in JAVA über die Typkonformität an die Typerweiterung gebunden ist und somit einer Variable eines Typs auch Objekte seiner Subtypen zugewiesen werden können, werden in C++ (wie auch in C#) Methoden zunächst einmal statisch gebunden. Das bedeutet im Klartext, dass auf einem Objekt immer die Methode aufgerufen wird, die in der Klasse definiert ist, deren Typ die Variable (und nicht das Objekt, auf das sie verweist) hat. Der tatsächliche Typ eines Objekts wird also ignoriert, es sei denn, die betreffende Methode wurde mit `virtual`⁷⁹ deklariert.

dynamisches Binden

Bei virtuellen Methoden wird hingegen wie in JAVA zur Laufzeit geprüft, welchen Typs das Objekt ist, und dann zur entsprechenden Methodenimplementierung verzweigt. Zu diesem Zweck hält das Laufzeitsystem eine sog. *Virtual function table*, in der die zum Objekt passende Implementierung nachgeschlagen werden kann. Diese Indirektion gilt jedoch als teuer (sie bedeutet einen Performanzverlust, den man schon an SMALLTALK immer bemängelt hatte) und soll daher nur wenn unbedingt notwendig durchgeführt werden. Folge ist, dass `virtual` (vor allem von SMALLTALK- und JAVA-Programmiererinnen) gelegentlich vergessen wird und Programme dann nicht wie erwartet funktionieren, oder dass die nachträgliche Erweiterung einer Klasse, auf die eine Programmiererin selbst keinen Einfluss hat, um Subklassen dazu führt, dass die Methoden der Subklasse auf Variablen der alten Klasse nicht aufgerufen werden können. In JAVA hat man deswegen bewusst davon Abstand genommen (und überlässt die Performanzsteigerung einem optimierenden Compiler); in C# hat man diese Entscheidung nicht nachvollzogen (s. Abschnitt 50.2).

In C++ wird Generizität mit Hilfe sog. **Templates** erreicht. Wie der Name schon nahelegt, ist ein Template ein Muster, anhand dessen neue, parameterlose Klassen erzeugt werden können. Im Gegensatz zu JAVA (und genau wie z. B. in ADA) werden aus Templates tatsächlich neue Klassen erzeugt: Man sagt, dass in C++ Typparameter *expandiert* werden. Das bedeutet, dass für jede Instanz eines generischen Typs (einer Template) ein neuer Typ tatsächlich erzeugt und kompiliert wird. Man kann sich den Mechanismus wie eine Textverarbeitung vorstellen, die das Template kopiert, alle Vorkommen der Typparameter darin durch tatsächliche Typen ersetzt, das ganze dann mit einem neuen Namen versieht und kompiliert. Tatsächlich wird die Generizität in C++ als ein Makro-Mechanismus angesehen; ihn umzusetzen ist die Aufgabe des Präprozessors.

Generizität

⁷⁹ Alternativ (und mit geringem Unterschied in der Bedeutung) kann auch `dynamic` verwendet werden.



1386 (**<Typ>**) **<Ausdruck>**

überzeugen den Compiler davon, dass das Objekt, für das **<Ausdruck>** steht, vom Typ **<Typ>** ist und entsprechend verwendet werden kann. Dabei wird nur leider vollkommen ignoriert, welchen Typs das Objekt tatsächlich ist, und ob dieser Typ zuweisungskompatibel mit **<Typ>** ist. Anders als in JAVA oder C# wird die Zulässigkeit dieser Typumwandlung auch nicht zur Laufzeit überprüft, ja sie kann zum Teil nicht einmal überprüft werden (s. Abschnitt 51.5.2) — wenn sie falsch war, dann hat man halt nicht richtig programmiert. So steht denn auch zu lesen:

Explicit type conversion is best avoided. Using a cast suppresses the type checking provided by the compiler and will therefore lead to surprises unless the programmer was right.

Margaret A. Ellis und Bjarne Stroustrup



Man spürt die Distanz der Autoren zu dem, was sie da beschreiben. Und so darf es als eine der großen Errungenschaften JAVAs gefeiert werden, dass es Type casts wenigstens zur Laufzeit auf Zulässigkeit prüft und damit ein Loch in der Typsicherheit schließt. Das führt uns zur dynamischen Seite des Typsystems von C++.

51.5.2 Dynamische Komponente

Die Beschreibung der dynamischen Typprüfung in C++ fällt knapp aus: Es gibt keine.

Nun wissen aber manche Objekte in C++ zumindest im Prinzip, von welcher Klasse sie Instanz sind, und zwar ganz einfach deswegen, weil sie einen Zeiger in die Sprungtabelle ihrer virtuellen Methoden besitzen. Da diese virtuelle Funktionstabelle für alle Objekte einer Klasse dieselbe ist, steht sie gewissermaßen für die Klasse. Und so wurde schließlich C++ doch noch eine Bibliotheksfunktion spendiert, die es erlaubt, für Objekte mit dynamisch gebundenen Methoden herauszufinden, Instanzen welcher Klassen sie sind.⁸⁰ Diese Information ist unter dem Namen *Runtime Type Information* (RTTI) bekannt.

Konkret kann die RTTI folgendermaßen ausgenutzt werden. Es gibt zunächst eine Funktion **typeid**, die man auf einer Referenz aufrufen kann.

Runtime Type
Information

Da die Funktion auf Klassennamen überladen ist und zudem eine Struktur zurückliefert, auf der **==** als Gleichheitstest definiert ist, lässt sich der Typ eines Objekts z. B. durch

⁸⁰ Aber leider eben nur für solche Objekte. Man kann freilich argumentieren, dass diese Information für andere Objekte auch nicht sonderlich interessant ist.



1387 `typeid(x) == typeid(T)`

prüfen (wobei `x` für eine Variable und `T` für eine Klasse stehen soll). Eine andere nützliche Funktion, die die RTTI ausnutzt, ist `dynamic_cast<T>(x)`; sie nimmt zwei Parameter, einen Typ (`T`) und ein Objekt (`x`), und liefert das Objekt mit dem Typ zurück, wenn die RTTI dies als typkorrekt erkennt, oder `0` sonst. In JAVA übersetzt hieße das:

1388 `(x instanceof T) ? (T) x : null`

wobei der Ausdruck `<A> ? : <C>` je nachdem, ob `<A>` zu wahr oder falsch auswertet, entweder das Ergebnis der Auswertung von `` oder von `<C>` zurückliefert (s. Kapitel 37 in Kurseinheit 4). `dynamic_cast<T>(x)` entspricht im wesentlichen `x as T` in C# (Abschnitt 50.4.4).

Selbsttestaufgabe 51.1

Schlagen Sie ein Verfahren vor, wie man mit Bordmitteln von C++ für Instanzen aller Klassen (und ohne Verwendung von `typeid`) herausfinden kann, Instanzen welcher Klasse sie sind.

51.6 Der ganze Rest

C++ ist eine sehr komplexe Sprache. Sie zu lernen dauert Jahre, selbst bei täglichem Umgang mit ihr; sie zu lehren erscheint fast aussichtslos. Selbst wenn man die zahlreichen Konstrukte der Sprache der Reihe nach durchginge, so ergibt sich die eigentliche Komplexität erst aus deren kombinierter Verwendung, und die Zahl der Möglichkeiten ist den Gesetzen der Kombinatorik folgend hoch. Entsprechend groß ist auch die Zahl der Idiome (Wendungen), die es für C++ gibt. Auch wenn längst nicht alle Beiträge zur Komplexität von C++ in Zusammenhang mit der Objektorientierung stehen, so ist es doch schwierig, sie davon zu trennen. Deshalb soll es an dieser Stelle auch genug damit sein.



52 EIFFEL

EIFFEL nimmt unter den hier behandelten Sprachen eine Sonderstellung ein. Es soll nämlich mehrere Dinge auf einmal sein:



- eine Sprache für objektorientierte Analyse und Design,
- eine Sprache für kommerzielle Programmierung und
- eine akademische Lehrsprache.

Das herausragende Merkmal, das EIFFEL zu dieser Multifunktion qualifiziert, ist die Integration von Zusicherungen (die Formulierung von Vorbedingungen, Nachbedingungen und Klasseninvarianten), die, als Verträge zwischen



dienstanbietenden und dienstnehmenden Klassen interpretiert, erlauben, das Was einer Software zumindest teilweise unabhängig vom Wie zu spezifizieren. Die Typsysteme, die Sie in den vorangegangenen Kapiteln kennengelernt haben und von denen auch EIFFEL eines besitzt, erlauben zwar auch schon, Zusicherungen auszudrücken, aber die sind jeweils auf die möglichen Werte einer Variable bezogen und bleiben dabei sowohl voneinander als auch von der Zeit unabhängig. EIFFEL erlaubt darüber hinaus, nahezu beliebige Bedingungen für Variablen- und Rückgabewerte von Methoden auszudrücken, die sowohl auf andere Werte als auch auf den zeitlichen Verlauf (vorher/nachher) Bezug nehmen können.

EIFFEL tritt in vielerlei Hinsicht in die Fußstapfen von PASCAL: Es ist nicht nur syntaktisch ähnlich, sondern ist auch um Sparsamkeit, Klarheit und Orthogonalität der Konzepte bemüht. Viele Dinge sind in EIFFEL ein klein bisschen anders als in anderen Sprachen, weswegen man meinen könnte, es sei aus Prinzip anders; die meisten Abweichungen sind aber wohl begründet und vermitteln mitunter eine angenehm andere Perspektive auf vertraute Dinge. So ist es eigentlich nur folgerichtig, dass der Erschaffer von EIFFEL, BERTRAND MEYER, Nachfolger von NIKLAUS WIRTH auf dessen Lehrstuhl an der ETH Zürich wurde.

akademische
Lehssprache

Es ist mir nicht ganz klar, warum EIFFEL kein größerer Erfolg beschieden ist — es mag zum einen an der über Jahre absolut unzureichenden Implementierung der Werkzeuge liegen (insbesondere des Compilers — es wurde anfangs noch nach C übersetzt; man male sich aus, welche Freude man als Programmiererin beim Debuggen hatte) und zum anderen an der Natur BERTRAND MEYERS, der sich mit seiner Kompromisslosigkeit nicht nur Freundinnen gemacht hat. Eine Rolle spielt sicher auch das unmögliche, aber trotzdem angestrebte Spagat zwischen kommerzieller Einsetzbarkeit und akademischer Eignung — viele Programmierweisen, die in der Praxis üblich sind und auf die kaum eine Entwicklerin verzichten will, sind akademisch verpönt. Das Typsystem EIFFELS schließlich trägt, wie Sie sich in Abschnitt 52.5 selbst vergewissern können, auch nicht unbedingt zur Akzeptanz bei. Auf der anderen Seite hat EIFFEL neben SMALLTALK noch am ehesten das Format, die Art, wie man über das Programmieren denkt, zu beeinflussen.

52.1 Das Programmiermodell EIFFELS

EIFFEL ist, wie alle anderen hier behandelten Sprachen mit Ausnahme von C++, eine rein objektorientierte Programmiersprache in dem Sinne, dass der gesamte Code innerhalb von Klassen angesiedelt ist. Klassen können (mit der in Abschnitt 52.5 gemachten Einschränkung) getrennt übersetzt werden. Dabei basiert die Ausführung von EIFFEL nicht auf einer virtuellen Maschine, sondern erfolgt direkt auf der Zielmaschine; allerdings gibt es inzwischen auch ein EIFFEL für .NET, das, wie alle .NET-Sprachen, zunächst in CIL und dann in Maschinencode übersetzt wird. Die Sprache sieht aufgrund ihrer Schlichtheit keine speziellen Konstrukte vor, die der Programmiererin erlauben, zwischen performanten und weniger performanten Implementierungen auszuwählen — mögliche Performanzgewinne werden ganz einem optimierenden Compiler überlassen. EIFFEL verwendet *Garbage collection* zur Speicherfreigabe.



52.2 Klassen als Module

Klassen sind in EIFFEL vor allem ein Mittel zur *Datenkapselung* („*Information hiding*“); alle Instanzvariablen oder Felder, in EIFFEL Attribute (engl. attributes) genannt, sind privat (weswegen man für sie auch keine Zugriffsmodifikatoren angeben kann). Um dennoch von außen darauf zugreifen zu können, benötigt man in EIFFEL (wie in SMALLTALK) zwingend Zugriffsmethoden, die allerdings (wie die Properties in C#) an der Aufrufstelle syntaktisch die Form von Variablen annehmen:

1389 `a.x := y`

ruft also eine Funktion (einen *Setter*) mit dem Parameter `y` auf,

1390 `y := a.x`

ruft eine auf, die einen (mit `y` zuweisungskompatiblen) Wert liefert (einen *Getter*). Funktion der Zugriffsmethoden ist es üblicherweise, eine entsprechende Instanzvariable zu setzen bzw. zu lesen; sie können aber auch etwas ganz anderes tun (vgl. dazu auch das Beispiel in Abschnitt 50.3.1). Insbesondere wird es dadurch möglich, den Zugriff auf Instanzvariablen mit Vor- und Nachbedingungen (s. Abschnitt 52.6) zu versehen. Auch bleibt der Programmiererin so die Freiheit, etwas, das wie eine Instanzvariable aussieht, nach außen anzubieten, ohne sich (dauerhaft) darauf festzulegen, dass es sich dabei auch tatsächlich um eine Instanzvariable handelt (die sog. *Repräsentationsunabhängigkeit*). Aber das hatten wir ja schon bei C# gesehen. Indexer gibt es in EIFFEL übrigens auch.

Die Methoden einer Klasse heißen in EIFFEL **Routinen** (routines) und werden logisch in zwei Gruppen unterteilt: **Abfragen** (queries) und **Befehle**

Routinen, Abfragen und Befehle

(commands). Abfragen geben über den Zustand von Objekten Auskunft, Befehle verändern ihn. Es ist in EIFFEL schlechter Stil (aber wird durch die Sprachdefinition nicht verhindert), dass eine Abfrage Seiteneffekte hat, also den Zustand des abgefragten Objekts (des Empfängers) oder eines anderen verändert.

Attribute (Instanzvariablen) und Routinen (Methoden) heißen in EIFFEL zusammen Features (entsprechend den Members in von C++ abgeleiteten Sprachen); sie sind in EIFFEL die einzigen Elemente einer Klassendefinition. Insbesondere ist es in EIFFEL nicht möglich, Klassendefinitionen zu schachteln (es gibt also keine inneren Klassen).

In EIFFEL gibt es keine Zugriffsmodifikatoren wie in JAVA/C#/C++: Wenn nichts weiter vermerkt wird, ist jedes Feature öffentlich zugänglich. Da Attribute jedoch nur über Zugriffsmethoden zugänglich sind, ist der Zustand (im Sinne der Belegung von Instanzvariablen; s. Kurseinheit 1, Kapitel 3) eines Objekts automatisch gekapselt: Man braucht ja schließlich keine Zugriffsmethoden zu spezifizieren und wenn man es doch tut, dann doch sicher nur, weil die entsprechenden Abfragen und Befehle Teil der Schnittstelle und kein Geheimnis sind.

Zugreifbarkeitsregeln



Nun wäre die Definition einer Schnittstelle (der Export von Features) in EIFFEL so aber reichlich unspezifisch — alle Klienten einer Klasse hätten (übrigens genau wie in SMALLTALK) das gleiche Bild von ihr. Das ist für größere Projekte kaum sinnvoll. Anstatt aber Zugreifbarkeit an Pakete zu binden (wie in JAVA und C#) oder an Freunde (*Friends*, wie in C++), bietet EIFFEL die Möglichkeit, einzelne Features *dediziert zu veröffentlichen* (zu exportieren), also unter Nennung der Klassen, die sie sehen können sollen. (Dies schließt die Nennung der leeren Menge ein, was dann bedeutet, dass keine andere Klasse diese Features sehen kann, also **private** in anderen Sprachen entspricht). Diese Klassen müssen dann umgekehrt die Features, die ihnen angeboten werden, nicht explizit importieren — das Wissen ob und der explizite Ausdruck der Abhängigkeit ist also in EIFFEL genau invers zu dem der anderen Sprachen. Dabei entspricht der dedizierte Export EIFFELS in etwa dem *qualifizierten Export* JAVAS, wir er dort allerdings nur für Module und nicht für Klassen definiert ist.

52.3 Anweisungen

Anweisungen sind in EIFFEL die Zuweisung, der Methodenaufruf, der Konstruktoraufruf (der Umgang mit Konstruktoren in EIFFEL unterscheidet sich erheblich von dem in SMALLTALK und auch von dem in JAVA/C#/C++; es wird hier jedoch nicht weiter darauf eingegangen) sowie die üblichen Kontrollstrukturen zur Verzweigung und der Wiederholung. EIFFEL hält sich dabei strikt an die Regeln der strukturierten Programmierung — jede Kontrollstruktur hat also genau einen Eingang und einen Ausgang. Das ist in der Praxis natürlich richtig lästig, führt zu ärgerlich langen Programmen und dürfte mit ein Grund für die mangelnde Akzeptanz EIFFELS unter professionellen Programmiererinnen sein. Davon unbeschadet bietet EIFFEL ein *Exception handling*, das sich angenehm von dem in JAVA/C#/C++ unterscheidet (es erlaubt insbesondere ein *Retry*, also ein Wiederholen eines fehlgeschlagenen Versuchs, das in anderen Sprachen durch Codermuster realisiert werden muss); auf Details von EIFFELS Exception handling wird hier jedoch nicht eingegangen, da es nur bedingt etwas mit Objektorientierung zu tun hat.

In EIFFEL ist, genau wie in PASCAL, das Semikolon Trennzeichen und nicht Teil einer Anweisung; darüber hinaus kann es am Ende einer Zeile auch weggelassen werden. Kleine EIFFEL-Programme wirken daher, und aufgrund des Verzichts auf Blöcke außerhalb von Kontrollstrukturen — es gibt weder `{...}` noch `begin ... end` —, optisch aufgeräumt; große Programme wirken jedoch schnell aufgeblättert.

52.4 Vererbung und Überladen

EIFFEL erlaubt Mehrfachvererbung. Es gestattet zudem sowohl das *Überschreiben* (in EIFFEL *Redefinition* genannt) als auch das Löschen von Methoden. Auf der Haben-Seite steht dabei, dass zu Beginn einer Klassendefinition deklariert werden muss, welche Methoden in der Klasse überschrieben (welche „Features redefiniert“) werden. Natürlich gibt es in EIFFEL auch *abstrakte Klassen*; das dazugehörige Schlüsselwort heißt jedoch **deferred** anstatt



abstract, ansonsten ist aber alles wie in JAVA (richtiger: in JAVA ist alles wie in EIFFEL — EIFFEL ist ca. zehn Jahre älter als JAVA).

Überladen und Umbenennung von geerbten Features

Link



BERTRAND MEYER ist ein bekennender Gegner des Überladens. Es ist in EIFFEL daher nicht erlaubt, dass eine Klasse zwei Methoden besitzt, die gleich heißen. Gleichwohl ist es erlaubt, dass verschiedene Klassen gleichnamige Methoden besitzen — nur dann dürfen diese Klassen nicht in Vererbungsbeziehung zueinander stehen, es sei denn, die Methoden besitzen gleiche oder kovariant redefinierte Parametertypen, denn dann sind die Methoden nicht überladen, sondern werden überschrieben (was dann allerdings durch eine Redefine-Deklaration anzudeuten ist). Nun kann man aber nicht erzwingen, dass zwei Klassen, von denen man erben möchte, keine gleichnamigen Methoden verwenden. Anstatt auf das Erben zu verzichten, ist es in EIFFEL deswegen möglich, geerbte Features umzubenennen. Es gibt zu diesem Zweck eine Rename-Klausel, die es erlaubt, zwei geerbte Features gleichen Namens mit unterschiedlichen Benennungen nebeneinander stehen zu lassen. Man beachte, dass das dynamische Binden davon unberührt bleibt: Über den Typ der Klasse, von der ein umbenanntes Feature geerbt wurde, angesprochen hört das Feature weiterhin auf seinen ursprünglichen Namen.

52.5 Das Typsystem EIFFELS

EIFFEL besitzt ein einheitliches Typsystem (keine separaten Referenz- und Werttypen) und unterscheidet auch nicht (wie JAVA) zwischen eingebauten Operatoren und programmiererrinnendefinierten Methoden: Die Operation **+** auf Integern beispielsweise ist (wie in SMALLTALK) nichts weiter als eine syntaktische Variante einer Methode **plus** definiert in einer Klasse **INTEGER** mit gleichem Parametertyp (EIFFEL verwendet per Konvention für Klassen und Typen vollständige Großschreibung; allerdings ist EIFFEL — wie PASCAL — nicht case sensitive). Die Einheitlichkeit des Typsystems von EIFFEL geht dabei über die von C# insofern hinaus, als es keine an bestimmte Typkonstruktoren gebundene Unterscheidung von Wert- und Referenztypen gibt; gleichzeitig unterscheidet es sich aber von dem SMALLTALKS (das sich ja zumindest dem Anschein nach vollständig auf Referenztypen festgelegt hat) insofern, als es auch Werttypen zulässt. Mehr dazu gleich.

Zu den prominentesten Eigenschaften des Typsystems von EIFFEL zählen

wichtige Eigenschaften

- Mehrfachvererbung,
- Generizität (genauer: beschränkter parametrischer Polymorphismus; s. Abschnitt 29.4 in Kurseinheit 3),
- das Unterdrücken von Instanzvariablen und Methoden in Subklassen (Löschen von Methoden; s. dazu auch Abschnitt 11.3 in Kurseinheit 1) sowie
- kovariante Redefinition, unterstützt durch sog. verankerte Typen (engl. anchored types; so gut wie ein Alleinstellungsmerkmal EIFFELS).



In gewisser Weise kann man das Unterdrücken von Methoden eines Typs in seinen Subtypen in EIFFEL als ein Spezialfall der kovarianten Redefinition ansehen, nämlich einen, in dem die Menge der möglichen Parameterobjekte auf die leere Menge eingeschränkt wird, so dass es keinen gültigen Aufruf gibt (vgl. Kapitel in Kurseinheit 3). Auch sind die Probleme, die kovariante Redefinition und Unterdrückung verursachen, ähnlich. Deshalb beschränken wir uns bei der Motivation (der Erklärung, warum EIFFEL über diese Eigenschaften verfügt), auf ein Beispiel für kovariante Redefinition.

52.5.1 Ein motivierendes Beispiel

MEYERS klassisches Beispiel für die Motivation kovarianter Redefinition und verankerter Typen à la EIFFEL soll Ihnen hier nicht vorenthalten werden — es wird immer wieder zitiert und ist, abgesehen davon, dass es zu Bemerkungen abseits der Informatik einlädt, intuitiv gut verständlich. Das Beispiel beginnt wie folgt:

```
1391 class SKIER feature
1392     roommate: SKIER;
1393         -- This skier's roommate
1394
1395     share (other: SKIER) is
1396         -- Choose other as roommate.
1397
1398     ...
1399     do
1400         roommate := other
1401     end
1402     ...
1403 end -- class SKIER
```

Die Idee ist, dass jugendliche Skifahrerinnen sich zu zweit ein Zimmer teilen. Wie Sie vielleicht schon selbst bemerkt haben, ist der durch die Klasse **Skier** definierte Typ rekursiv: Seine Instanzvariable **roommate** ist selbst vom Typ **Skier**. Somit wird z. B. der folgende Aufruf von **share** möglich (typkorrekt):

```
1402 s1, s2 : SKIER
1403 s1.share(s2)
```

Nun ist das Beispiel so noch unvollständig. Es ergibt sich nämlich aus der Sache, dass die jungen Skifahrerinnen nach Geschlechtern getrennt untergebracht werden sollen. Dazu kann man in EIFFEL (dank erlaubter kovarianter Redefinition) einfach schreiben:

```
1404 class GIRL inherit
1405     SKIER
1406     redefine roommate, share end
1407 feature
1408     roommate : GIRL;
1409
1410     share (other : GIRL) is
1411     ...
1412     do
1413         roommate := other
1414     end
```



```
1414 end -- class GIRL
```

sowie entsprechend für Jungen

```
1415 class BOY inherit
1416   SKIER
1417     redefine roommate, share end
1418 feature
1419   roommate : BOY;
1420   share (other : BOY) is
1421     ...
1422     do
1423       roommate := other
1424     end
1425 end -- class BOY
```

Zwei Dinge fallen auf: Zum einen unterscheiden sich die beiden Subklassen lediglich darin, dass sie `roommate` einen anderen Typ zuordnen und dies in der Folge auch für den Parameter von `share` tun (müssen), zum anderen sind, wie bei Kovarianz üblich, bei den Variablen Deklarationen

```
1426 s1: SKIER; b1: BOY; g1: GIRL
```

die nachfolgenden Anweisungen nach den Regeln der Zuweisungskompatibilität zulässig:

```
1427 s1 := b1;
1428 s1.share(g1)
```

Beim Funktionsaufruf von `share` wird nun aber, dank dynamischer Bindung, `g1` (und damit ein Objekt vom Typ `GIRL`) an `roommate` in `BOY` und damit an eine Variable vom Typ `BOY` zugewiesen. Schon ist das Programm nicht mehr typkorrekt.

Naturgemäß kommt in EIFFEL dem Versuch der Reparatur des durch obiges Beispiel angedeuteten Verlusts der Typkorrektheit eine besondere Bedeutung zu. Hier sei nur noch schnell (und ohne etwas von der Lösung vorwegzunehmen) erwähnt, dass der eigentliche Fehler in der implizit angenommenen Allquantifizierung einer Deklaration wie `SKIER.share(SKIER)` (oder, mathematisch ausgedrückt, $share: SKIER \times SKIER$) liegt: Wie schon in Kurseinheit 3, Kapitel 26 erwähnt, bedeutet eine solche Deklaration eben *nicht*, dass *alle* Skifahrerinnen (beiden Geschlechts) ihr Zimmer mit *allen* Skifahrerinnen (wieder beiden Geschlechts) teilen können. Das ist aber auch schon unabhängig von den möglichen Geschlechtern (Subklassen) nicht der Fall: Wenn eine Skifahrerin beispielsweise (vorübergehend) ansteckend erkrankt ist, kann sie höchstens mit anderen Kranken das Zimmer teilen. Eine Klassifikation nach Kranken und Gesunden entzieht sich aber, da sie nicht dauerhaft ist, den Möglichkeiten der statischen Typisierung (zumindest den offensichtlichen).

Ursache des Verlusts der Typkorrektheit



52.5.2 Statische Komponente

Wie auch in JAVA definiert in EIFFEL jede unparametrisierte Klasse einen Typ und jede parametisierte Klasse eine (generische) Menge von Typen. Alle Variablen (inkl. Methoden⁸¹ und deren Parameter) müssen einen Typ haben. *Zuweisungskompatibilität* ist an nominale *Typkonformität* gebunden, die wiederum mit der *Typerweiterung* (in EIFFEL einfach *Vererbung* genannt) einhergeht — ganz wie in JAVA. Anders als in JAVA ist es jedoch zulässig, Instanzvariablen und Funktionsparameter wie im obigen Beispiel kovariant zu redefinieren — von kontravarianter Redefinition will MEYER nichts wissen (eine Begründung sollten Sie mittlerweile selbst zur Verfügung haben). Das bedeutet allerdings mangelnde *Substituierbarkeit* und bereitet EIFFEL erwartungsgemäß einige nichttriviale Probleme.

Zunächst einmal wollen wir uns das Typsystem EIFFELS noch etwas genauer ansehen. Es basiert, wie in der objektorientierten Programmierung üblich, auf dem Begriff der Typkonformität (Kapitel 23 in Kurseinheit 3). In EIFFEL ist ein Typ U typkonform zu einem Typ T

Bedingungen der Typkonformität

- wenn U und T gleich sind,
- wenn U eine direkte Erweiterung von T ist (direkt von T erbt) und wenn zusätzlich, im Falle von parametrischer Erzeugung von U und T, jeder tatsächliche Typparameter von U konform ist zum entsprechenden tatsächlichen Typparameter von T oder
- wenn es einen Typ V gibt, so dass U typkonform mit V und V typkonform mit T ist (U ist eine indirekte Erweiterung von T).

Außerdem gibt es in EIFFEL noch einen Typkonstruktor `like <ein Ausdruck>` (s. u.), dessen erzeugter Typ typkonform zum Typ von `<ein Ausdruck>` ist. Dieser spielt bei der kovarianten Redefinition eine wichtige Rolle. Zu einem so erzeugten Typ ist jedoch nur der Basistyp konform, keiner seiner Subtypen.

Dazu zunächst ein paar Beispiele. Bei

```
1429 class B inherit A ...
1430 a : A
1431 b : B
```

ist

```
1432 a := b
```

OK,

```
1433 b := a
```

⁸¹ Mit dem Typ einer Methode ist hier der Rückgabetyp gemeint.



hingegen nicht. Im Falle der Vererbung bei parametrischem Polymorphismus wie in

```
1434 class GSub[B] inherit class GSUPER[B]  
1435 x : GSub[integer]  
1436 y : GSUPER[integer]
```

ist wieder

```
1437 y := x
```

zuweisungskompatibel, die umgekehrte Zuweisung jedoch nicht. Beschränkter parametrischer Polymorphismus wird in EIFFEL übrigens wie folgt notiert:

```
1438 class SortedList[G -> Comparable]
```

EIFFEL benutzt also eckige Klammern und \rightarrow anstelle von spitzen Klammern und `extends` in JAVA.

In EIFFEL wird übrigens anders als in C# nicht pro Typkonstruktor zwischen Wert- und Referenztyp unterschieden — zu jedem kann es (ähnlich wie in C++) beide Formen geben. Dazu gibt es in EIFFEL die Möglichkeit, bei einer Deklaration anzugeben, dass Variablen eines Typs Wertsemantik, also ein Objekt anstelle einer Referenz auf ein Objekt zum Inhalt haben sollen (s. Kurseinheit 1, Abschnitt 1.5). Dies ist manchmal für alle Variablen eines Typs sinnvoll (z. B. bei Zahlen und Wahrheitswerten), manchmal aber auch nur für manche. Und so gibt es in EIFFEL einen Typkonstruktor `expanded`, der, in Variablen-deklaration wie in

Typkonstruktoren für Referenz- und Wertsemantik

```
1439 x : expanded C
```

eingesetzt, einer einzelnen Variable eine Wertsemantik gibt, und der in Klassendefinitionen wie in

```
1440 expanded class C ...
```

verwendet allen Variablen des entsprechenden Typs (der entsprechenden Typen im Falle einer generischen Typdefinition) automatisch Wertsemantik gibt. EIFFELS Typkonstruktor `expanded` entspricht also gewissermaßen einer Umkehrung des in PASCAL-artigen Sprachen verwendeten Typkonstruktors `^` (Zeiger auf): Wenn `expanded` *nicht* verwendet wird, handelt es sich um einen Zeigertypen.

Die Unterscheidung von Wert- und Referenztypen einer Klasse hat in EIFFEL einen starken konzeptuellen Hintergrund: Sie unterstützt die bereits in Abschnitt 2.3 diskutierte Komposition und ihre Abgrenzung als eine besondere Beziehung zwischen Objekten, nämlich der, die das Enthaltensein von Objekten in anderen ausdrückt (vgl. dazu auch Kapitel 59 in Kurseinheit 6). Nun ist es in der Realität so, dass nicht alle Instanzen einer Klasse immer

Hintergrund des gleichzeitigen Angebots von Wert- und Referenzsemantik



entweder Komponenten (also in anderen Objekten enthalten) oder freie Objekte (also nirgends enthalten) sind. EIFFEL wird dem gerecht, indem es erlaubt, von einer Klasse fallweise Komponentenobjekte (über **expanded** Variablen) und freie Objekte (über normale Variablen) zu haben. Dieses Feature ist nicht in C# (zumindest nicht im Safe mode) zu haben, denn dort definieren Structs ausschließlich Werttypen und Klassen ausschließlich Referenztypen, und schon gar nicht in SMALLTALK oder JAVA — in C++ (und im Unsafe mode von C#) kann man es simulieren, zahlt dafür aber den Preis, mit expliziten Pointern hantieren zu müssen, was nach landläufiger Auffassung ein ziemlich hoher ist.

Nun stand bereits in Kurseinheit 1, Abschnitt 1.6 zu lesen, dass bei der Zuweisung zwischen zwei Variablen mit Wertsemantik der Wert der einen Variable in die andere Variable kopiert wird, während bei der Zuweisung zwischen zwei Variablen mit Referenzsemantik lediglich der Zeiger kopiert wird. Dies ist auch in EIFFEL so. Bei der Zuweisung einer Variable mit Referenzsemantik an eine Variable mit Wertsemantik reicht es jedoch nicht, einen Zeiger zu kopieren, denn die Zielvariable hat keinen Platz für einen Zeiger, sondern für die Attributwerte — stattdessen wird hier das Objekt, auf das der Zeiger verweist, kopiert (genauer: es werden die Attribute des Objekts in den für die Attribute des Werts reservierten Speicher der Variable kopiert). Im umgekehrten Fall, also wenn eine Variable mit Wertsemantik an eine Variable mit Referenzsemantik zugewiesen wird, könnte man annehmen, dass ein Zeiger auf den Wert erzeugt und zugewiesen wird; dies würde aber bedeuten, dass dadurch ein *Alias* auf einen Wert entstünde, was nicht der Semantik der Komposition (*Aggregation*; s. Abschnitt 2.3 in Kurseinheit 1) entspräche. Was stattdessen passiert, ist, dass ein Klon des Objekts erzeugt wird und eine Referenz auf diesen Klon übergeben wird. Dies ist eine äußerst sinnvolle Festlegung.

Zuweisungen zwischen Variablen unterschiedlicher Semantik

Verankerte Typdeklarationen haben in EIFFEL die Form

verankerte Typen

1441 **x : like y**

wobei **y** ein bereits typisiertes Programmelement (also z. B. eine Instanzvariable) ist. Eine solche Verankerung bewirkt, dass sich der Typ von **x** automatisch mit dem von **y** verändert. Dies hat zunächst noch nichts mit Kovarianz zu tun, wie das folgende Beispiel zeigt.

Anstatt in EIFFEL

```
1442 class A
1443   feature
1444     f : X
1445     setF(v : X)
1446     getF() : X
1447   end
1448 ...
1449 end

1450 class B
1451   inherit A
1452   redefine f
```



```

1453 feature
1454   f : Y
1455   setF(v : Y)
1456   getF() : Y
1457 end
1458 ...
1459 end

```

zu schreiben, d. h., bei Veränderung des Typs der Instanzvariable **f** von X zu Y alle Parametertypen, die davon berührt sind, mit zu verändern, reicht es aus,

```

1460 class A
1461   feature
1462     f : X
1463     setF(v : like f)
1464     getF() : like f
1465   end
1466 ...
1467 end

1468 class B
1469   inherit A
1470   redefine f
1471   feature
1472     f : Y
1473   end
1474 ...
1475 end

```

zu schreiben. Da die Typen der Parameter von **setF** und **getF** alle per Deklaration dieselben sind wie der Typ von **f**, muss in der Definition von B textuell nichts anderes stehen. Da nun aber in EIFFEL die Redefinition von Instanzvariablen per Definition immer kovariant sein muss, muss Y ein Subtyp von X sein. Der Rückgabetyp von **getF** und der Parametertyp von **setF** ändern sich damit automatisch ebenfalls kovariant.

Ein besonderer Fall von verankerten Typen ergibt sich bei rekursiven Typen, also Typen, deren Definition den definierten Typ selbst referenziert:

Verankerung bei rekursiven Typen

In diesem Fall schreibt man in EIFFEL anstelle der Typpreferenz bei der Deklaration einer Variable vom zu definierenden Typ **like Current**. Bei einem entsprechend deklarierten Feld ändert sich der Typ bei der Vererbung also immer automatisch zum erbenden Typ hin ab, also immer mit dem Typ und damit kovariant. Für das Beispiel der zu trennenden Skifahrerinnen (Zeilen 1391–1425) ergibt sich damit

```

1476 class SKIER feature
1477   roommate: like Current;

1478   share (other: like Current) is
1479   ...
1480   do
1481     roommate := other
1482   end
1483 ...
1484 end -- class SKIER

```



Die beiden erbenden Klassen GIRL und BOY müssen dann nichts mehr redefinieren.

Wir können nun zur Lösung des Problems der Kovarianz in EIFFEL kommen. Die obige Konstruktion

Lösung des Problems der Kovarianz

```
1485 s1 : SKIER; b1 : BOY; g1 : GIRL  
1486 s1 := b1;  
1487 s1.share(g1)
```

(hier unverändert wiederholt) wird dann vom Type checker zur Übersetzungszeit zurückgewiesen, da g1 nicht vom Typ **like s1** ist, was aber laut Typkonformitätsregeln von EIFFEL notwendig wäre. Leider ist das nur ein Teilerfolg.

Es ist nämlich andersherum **like s1** mit dem Typ von s1 konform. Und so wird es möglich, dass bei zusätzlicher Deklaration von

```
1488 g2 : like s1
```

(wobei der Typ von s1 ja SKIER ist)

```
1489 s1 := g1; g2 := s1
```

und anschließend

```
1490 s1 := b1
```

doch wieder

```
1491 s1.share(g2)
```

zulässig ist und damit ein Mädchen einem Jungen ins Zimmer gesteckt wird.

Die erste und einfachste Möglichkeit, dies zu verhindern, wäre, die Zuweisungskompatibilität für Variablen mit verankerten Typen und Typkern einzuschränken und unter ihnen nur noch Typäquivalenz zu akzeptieren. Es wären dann nur noch Zuweisungen zwischen Variablen eines als Typanker verwendeten Typs mit solchen, die ihn als Anker benutzen, erlaubt; insbesondere wäre eine Zuweisung wie s1 := g1 (die ja notwendig war, um ein Objekt vom Typ GIRL so zu „verpacken“, dass es an g2 mit seinem verankerten Typ **like s1** zugewiesen werden kann) damit nicht mehr möglich. Alle anderen Zuweisungen wären natürlich weiter zugelassen; dies hätte jedoch zur Konsequenz, dass bei verankert genutzten Typen geschlossene Zirkel entstünden, aus denen kein Objekt hinaus und in die keines hinein käme (außer bei seiner Erzeugung). Außerdem wäre es für die Programmiererin schwer, vorab zu entscheiden, ob ein Typ entweder als Anker zur kovarianten Redefinition verwendet oder ob er polymorph, also für Variablen, die Objekte unterschiedlichen Typs haben dürfen, genutzt werden soll.

Lösungs- möglichkeiten



Die zweite Möglichkeit wäre, eine Typinferenz für das gesamte Programm durchzuführen, um die möglichen Zuweisungen an Variablen zu sammeln. Dazu sind insbesondere alle Methodenaufrufe anzusehen (Zuweisungen an Instanzvariablen sind in EIFFEL nur innerhalb einer Klasse erlaubt) und diese können je nach Konfiguration des endgültigen Systems sehr unterschiedlich ausfallen. Eine solche Typinferenz ist aber in den meisten Fällen unrealistisch.

Die dritte Möglichkeit ist die, alle dynamisch gebundenen Aufrufe von Methoden, deren Verfügbarkeit oder Parametertypen sich in überschriebenen Versionen („Redefinitionen“) ändern (die von MEYER so genannten CAT-Calls, wobei CAT für „Change Availability or Type“ steht), zu verbieten. Ein Aufruf von `share` auf `s1` wie oben ist damit verboten, weil `share` in BOY und GIRL kovariant redefiniert wird. Auf einer Variable vom Typ GIRL oder BOY wäre er hingegen zulässig, solange sichergestellt ist, dass diese Variable keinen Wert von einem Subtyp von GIRL bzw. BOY zugewiesen bekommen kann. Das ist möglich, wenn kein solcher Subtyp existiert oder wenn keine Zuweisung an die Variable existiert, bei der die rechte Seite ein Subtyp der Variable ist. Das erste ist lokal nicht nachzuweisen, das zweite hingegen schon, jedoch nur für *explizite Zuweisungen* (inkl. der Instanziierung, die in EIFFEL auf einer Variable durchgeführt wird und ihr automatisch einen Wert gibt). Für die Zuweisung an formale Parameter kann dies jedoch nicht lokal nachgewiesen werden, weil im Gegensatz zu expliziten Zuweisungen an Variablen die Methodenaufrufe von überall her erfolgen können.

Wie Sie sehen, sind die Bedingungen ziemlich restriktiv, und man kommt nicht umhin, das Typsystem von EIFFEL als etwas eigenartig zu empfinden. Wie es sich in der Praxis auswirkt, ist mir leider nicht bekannt; MEYER behauptet, dass die Probleme praktisch keine Rolle spielen. Ich möchte hinzufügen, dass falls doch, die durchschnittliche Programmiererin kaum verstehen wird, was denn nun genau das Problem ist und was sie tun kann, es zu umgehen.

52.5.3 Die dynamische Komponente

Bei allen Bemühungen, für EIFFEL ein möglichst „wasserdichtes“ Typsystem vorzulegen und dabei so viel wie möglich zur Übersetzungszeit zu erledigen, bleibt es natürlich auch in EIFFEL-Programmen nicht aus, dass man in einen Container (eine Variable oder eine Collection) Elemente ungleichen Typs hineinpackt und hinterher wissen will, welchen genauen Typs ein Element ist, um es seinem Typ entsprechend verwenden zu können. Nicht immer wird man die dazu notwendige *Fallunterscheidung* dem dynamischen Binden (einem dynamisch gebundenen Methodenaufruf) überlassen wollen; manchmal ist es einfach einfacher (und besser nachvollziehbar), wenn man den Typ explizit prüft und innerhalb einer Methode entsprechend verzweigt.

Solche Typtests werden in EIFFEL von einem sog. *Zuweisungsversuch* (engl. assignment attempt) übernommen, der bei mangelnder (dynamisch festgestellter) Zuweisungskompatibilität einfach `void` (das Äquivalent von `nil` in SMALLTALK und `null` in JAVA) zuweist:

Typtests und
Typumwandlungen



ergibt nie einen Typfehler, sondern führt höchstens dazu, dass **a** **void** zugewiesen wird. Es bleibt dann die Aufgabe der Programmiererin, **a** nach der Zuweisung zu kontrollieren. Es entspricht dies direkt dem **as** aus C#, dem **dynamic_cast<T>(x)** aus C++ sowie dem JAVA-Konstrukt

```
1493 if (b instanceof A) a = (A) b; else a = null;
```

oder auch kryptischer

```
1494 a = (b instanceof A) ? (A) b : null;
```

wobei **A** der Typ von **a** sei (man beachte das ärgerliche, aber in C-artigen Sprachen notwendige Semikolon vor dem Else). Explizite Type casts gibt es in EIFFEL nicht; sie können also auch keine Laufzeitfehler verursachen. Der Zuweisungsversuch erfüllt aber weitgehend die Funktion einer Typumwandlung, denn er ist nur erfolgreich, wenn die rechte Seite zuweisungskompatibel mit der linken ist, was per Definition nur dann der Fall ist, wenn die rechte Seite ein Objekt eines Subtyps (einschließlich Gleichheit) der linken Seite hat. Es wird hier allerdings die Typumwandlung immer mit einer Zuweisung verbunden; man braucht also u. U. eine temporäre Variable, die man sich sonst hätte sparen können. Dass der Zuweisungsversuch in EIFFEL anders als der Down cast in JAVA keinen Laufzeitfehler verursachen kann, ist wenig tröstlich, denn der Wert **void** in einer Variable kann es natürlich schon; in Wirklichkeit wird hier lediglich ein Type cast error gegen eine Null pointer exception getauscht.

Die Typumwandlung wird in EIFFEL aber auch noch für etwas anderes gebraucht, nämlich für das Binden von Aufrufen kovariant redefinierter Methoden. Da EIFFEL ja, wie oben beschrieben, polymorphe CAT-Calls verbieten muss, diese aber gleichwohl notwendig sein können, hat man nur die Möglichkeit, die dynamische Bindung programmatisch zu emulieren. Und dafür braucht man Zuweisungsversuche, wie folgendes Beispiel zeigt:

emulierte
dynamische Bindung

```
1495 g : GIRL; b : BOY
1496 g ?= s1;
1497 if g /= Void then
1498   g.share(...)
1499 end else
1500   b ?= s1;
1501   if b /= Void then
1502     b.share(...)
1503   end
1504 end
```

Nun ja.



52.6 Zusicherungen in EIFFEL: Vorbedingungen, Nachbedingungen und Klasseninvarianten

Praktisch ein Alleinstellungsmerkmal EIFFELS ist die Integration von Zusicherungen in Form von Vor- und Nachbedingungen bei Methodenaufrufen. Bei der Behandlung JAVAs war uns ja schon die Assert-Anweisung begegnet, die es erlaubte, Zusicherungen zur Laufzeit auszuwerten und das Programm ggf., bei einer Verletzung, abzubrechen. Da es sich aber um eine Anweisung handelte, gab es keine von der Sprachdefinition vorgesehenen Orte, an denen solche Zusicherungen auftreten sollten — sie an passenden Stellen einzustreuen war ganz der Programmiererin überlassen. In EIFFEL ist das anders.

EIFFELS Syntax zur Definition einer Methode sieht zwei Schlüsselwörter, **require** und **ensure**, vor, von denen das erste vor der Definition der Implementierung der Methode (dem Methodenrumpf), das zweite danach auftreten kann. Beiden Schlüsselwörtern folgen können Boolesche Ausdrücke, die allesamt zu „wahr“ auswerten müssen. Die Idee hinter einer Require-Klausel ist, dass, damit die betreffende Methode richtig funktionieren kann, die darin ausgedrückten Bedingungen erfüllt sein müssen. So ist es beispielsweise sinnvoll, für die Methode **pop** der Klasse **STACK** zu verlangen, dass der betreffende Stack, auf dem die Methode aufgerufen wird, nicht leer ist. Die Idee hinter einer Ensure-Klausel ist, dass eine Methode, deren Require-Klausel erfüllt war, im Gegenzug garantieren muss, dass sie die in der Ensure-Klausel ausgedrückten Bedingungen erfüllt. Im Beispiel des Stacks wäre das beispielsweise, dass nach einem **push** das übergebene Element auch tatsächlich oben auf dem Stapel liegt, die Methode **top** also beispielsweise das soeben auf den Stapel gelegte Element liefert. Um ihre Bedingungen zu formulieren, dürfen die Ausdrücke in beiden Klauseln auf Abfragen (queries) der Klasse zurückgreifen. Diese sollten dazu aber tunlichst nebeneffektfrei sein, zum einen, weil die Ensure-Klausel sonst nicht garantieren kann, dass eine geprüfte Bedingung auch nach ihrer vollständigen Abarbeitung immer noch wahr ist (man bedenke nur, was wäre, wenn die Ensure-Klausel zu **push** die Methode **pop** aufrufen würde!), zum anderen aber auch, weil auch in EIFFEL (wie in JAVA) die Überprüfung der Zusicherungen zur Laufzeit abgestellt werden kann (weswegen dann das Programm mit Überprüfung der Zusicherungen eine andere Semantik hätte als ohne; vgl. die Anmerkungen zu JAVAs Assert-Anweisung in Kurseinheit 4, Kapitel 38).

**Vor- und
Nachbedingungen:
requires und
ensures**

Neben der Möglichkeit, Vor- und Nachbedingungen zu formulieren, gibt es in EIFFEL noch die Möglichkeit, sog. Klasseninvarianten (Schlüsselwort **invariant**) zu deklarieren. Klasseninvarianten müssen jederzeit zwischen zwei Methodenaufrufen gelten; man kann sich vorstellen, dass sie jeder Vor- und Nachbedingung per Konjunktion hinzugefügt werden. Auf die etwas subtilen Probleme, die das Aliasing in Zusammenhang mit Zusicherungen schafft, wollen wir an dieser Stelle nicht eingehen; Kurs 01853 befasst sich ausführlicher damit.

**Klasseninvarianten:
invariant**

Zusicherungen werden in EIFFEL von Klassen auf ihre Subklassen vererbt. Wenn dabei eine Methode redefiniert wird, dann dürfen auch Vor- und

**Zusicherungen und
Vererbung**



Kurs

Nachbedingung angepasst werden. Allerdings gilt hier, dass die Vorbedingung nur aufgeweicht, die Nachbedingung nur verschärft werden darf. Die Sprachdefinition EIFFELS stellt die Einhaltung dieser Bedingung automatisch sicher, indem die Vorbedingung einer redefinierten Methode mit der geerbten Vorbedingung implizit disjunktiv und die redefinierte Nachbedingung mit der geerbten implizit konjunktiv verknüpft wird. Mehr zu diesem Thema können Sie ebenfalls Kurs 01853 entnehmen.

Selbsttestaufgabe 52.1

Begründen Sie die eben geschilderten Verknüpfungen von geerbten und redefinierten Zusicherungen. Können Sie einen Zusammenhang zu der Veränderung von Parametertypen beim Subtyping herstellen? Fällt Ihnen etwas auf?

52.7 Tupel anstelle von Klassen

Ein relativ neues Feature von EIFFEL sind die sog. **Tupel**. Tupel erlauben es, ohne großen Aufwand mehrere Objekte zu einem zu gruppieren. Insbesondere ist dafür keine Definition einer Klasse notwendig. Tupel sind also vor allem da interessant, wo strukturierte Daten vorkommen, ohne dass man der Struktur eine eigenständige Bedeutung beimessen würde. Dies ist am prominentesten bei Methoden der Fall, die mehrere anstatt eines Wertes zurückgeben sollen.

Tupeltypen werden deklariert, indem man für jede Stelle einen Typ angibt; konkrete Werte eines Tupels werden in eckigen Klammern notiert:

1505 [1, a.x, "abc"]

ist ein Tupelausdruck. Die Stellen eines Tupels können auch mit Namen versehen werden:

1506 [a: 1, b: a.x, c: "abc"]

ist ein solches Tupel, bei dem die Stellen „a“, „b“ und „c“ heißen. Da Tupeltypen unbenannt sind, erfolgt die Zuordnung eines Tupels zu einem Tupeltyp (die ja normalerweise bei Instanziierung erfolgt) aufgrund eines Abgleichs der vorgefundenen mit den deklarierten Stellentypen (sowie ggf. der Namen der Stellen).

52.8 Fazit

Zum Thema objektorientierte Programmiersprachen ist viel geschrieben worden. Eines der beeindruckendsten Werke ist sicher das Buch „Object-Oriented Software Construction“ von BERTRAND MEYER: Auch wenn er darin letzten Endes versucht, der Leserin EIFFEL zu verkaufen, und selbst wenn er in Sachen Typsystem einen gewissen Starrsinn an den Tag legt, so steckt das Buch doch voller zeitloser Weisheiten, die sich allesamt in der — seiner — Sprache EIFFEL widerfinden. Eine unbedingte Leseempfehlung!



53 Lösungen zu den Selbsttestaufgaben

Selbsttestaufgabe 50.1 (Seite 258)

Die Regel für die Zuweisungskompatibilität an der Aufrufstelle, also die Zuweisung des tatsächlichen an den formalen Parameter, muss auf Typäquivalenz eingeschränkt werden, da sonst (im Rumpf der aufgerufenen Methode) dem tatsächlichen Parameter ein Wert eines Supertyps des Typs des tatsächlichen Parameters zugewiesen werden kann. Praktisch heißt das für C#, dass der tatsächliche und der formale Parameter vom gleichen Typ sein müssen.

Selbsttestaufgabe 51.1 (Seite 276)

Man deklariert eine virtuelle Methode `getClass()` in einer abstrakten Klasse (nennen wir sie `Object`), lässt jede Klasse, die die Funktionalität haben soll, von dieser Klasse ableiten und über- schreibt `getClass()` in diesen Klassen mit der Rückgabe der jeweiligen Klasse bzw. eines Repräsentanten der Klasse.

Selbsttestaufgabe 52.1 (Seite 291)

Die Konjunktion (logische Und-Verknüpfung) einer Bedingung mit einer beliebigen anderen entspricht einer Verschärfung der Bedingung (sie ist in weniger Fällen wahr), die Disjunktion (logische Oder-Verknüpfung) einer Aufweichung (sie ist in mehr Fällen wahr).

Typinvarianten sind spezielle, Variablenwerte betreffende Zusicherungen. Typinvarianten auf Eingabeparametern entsprechen Vorbedingungen, solche auf Ausgabeparametern (dazu zählt der Rückgabeparameter) entsprechen Nachbedingungen. Die Aufweichung der Vorbedingungen in Subklassen entspricht der Kontravarianz von Eingabeparametern, die Verschärfung der Nachbedingungen der Kovarianz der Ergebnistypen.

Bemerkenswerterweise verlangt EIFFEL bei den Vorbedingungen Kontravarianz, bei den Eingabeparametern jedoch Kovarianz.



Kurseinheit 6: Probleme der objektorientierten Programmierung

In den vorangegangenen Kurseinheiten wurde bereits an verschiedenen Stellen auf Probleme der objektorientierten Programmierung hingewiesen. Diese sollen nun in dieser Kurseinheit zusammengefasst dargestellt werden.

54 Das Problem der Substituierbarkeit

In Kapitel 26 sind wir ja schon ausführlicher auf den Begriff des *Subtyping*s eingegangen. Das Subtyping sollte *Zuweisungskompatibilität* zwischen verschiedenen Typen gestatten, also erlauben, dass Objekte eines Typs Variablen eines anderen Typs, nämlich eines Supertyps, zugewiesen werden. Das führt nun zu dem Problem, dass man — aufgrund des *dynamischen Bindens* von Methodenaufrufen — bei Vorliegen des Programmfragments

```
1507 | e <T> |
1508 e := <ein Ausdruck>.
1509 e m
```

selbst bei Kenntnis des Typs T nicht sagen kann, welchen Effekt der Aufruf von Methode *m* in Zeile 1509 hat.⁸² Nach den Regeln gängiger objektorientierter Programmiersprachen zur Zuweisungskompatibilität weiß man lediglich, dass es sich beim Typ des Empfängerobjekts *e* um einen Subtyp von T handeln muss⁸³ — man weiß aber nicht, um welchen. Bei statischer, lokaler Betrachtung⁸⁴, also bei mangelnder Kenntnis des Typs des von *e* bezeichneten Objekts sowie aller infrage kommenden Subtypen von T, tappt man hier vollkommen im Dunkeln. Da die Erweiterung um Subklassen bzw. Subtypen aber gerade eine der Errungenschaften der objektorientierten Programmierung ist, hat man es dabei mit einem echten Problem zu tun.

Das Problem lässt sich als ein Problem der formalen Programmverifikation ausdrücken: Wie lässt sich

⁸² Die Parameter des Methodenaufrufs sind hier unwichtig, da wir davon ausgehen, dass Methodenaufrufe nur unter Berücksichtigung des Empfängerobjekts gebunden werden.

⁸³ In typlosen Programmiersprachen wie SMALLTALK weiß man nicht einmal das.

⁸⁴ Zur Wichtigkeit von lokaler Betrachtung s. a. Abschnitt 56.



also dass bei Vorliegen der Bedingung P vor Ausführung des Methodenaufrufs $e \rightarrow m$ nach dessen Ausführung die Bedingung Q eingehalten wird, beweisen? Ein solcher Beweis verlangt immerhin genaue Kenntnis davon, was der Methodenaufruf tut, oder vielmehr, welchen Effekt dieses Tun hat. Dazu muss man aber die Implementierung der Methode kennen.

Umgekehrt ist es für die Pflege und Weiterentwicklung eines Programms wichtig zu wissen, welchen Bedingungen die Methoden einer in ein Programm neu eingeführten Klasse genügen müssen, damit das Programm auch hinterher noch funktioniert. Wenn die neue Implementierung sich in den Kanon der bereits bestehenden einordnet, ohne ein unerwartetes Verhalten einzubringen, wenn sie also für alle Aufrufe gemachte Zusicherungen der Form von (54.1) einhält, dann ist das Funktionieren nicht gefährdet — andernfalls hingegen schon. Nur um das zu garantieren, müssen die Bedingungen bekannt sein.

Es stellt sich also die Frage, wie man den Effekt aller Implementierungen von m für Subtypen von T (einschließlich T selbst) fassen kann. Eine naive Beantwortung der Frage würde vorschlagen, dass man sich alle diese Implementierungen ansieht und auf dieser Basis eine *Fallunterscheidung* präsentiert: Wenn das Objekt e vom Typ T ist, dann hat m den und den Effekt, wenn es von Subtyp T_1 ist, dann hat m den und den Effekt usw. Die Zusammenfassung dieser Fallunterscheidungen würde dann alle Effekte mit logisch Oder verknüpfen. Diese Möglichkeit hat jedoch neben ihrer unnötigen Sperrigkeit den Nachteil, dass dabei noch gar nicht vorhandene Implementierungen unberücksichtigt bleiben müssen. Was man stattdessen möchte, ist die Gewissheit, dass eine lokale, „modulare“ Betrachtung ausreicht und man nicht jedes Mal eine Analyse des gesamten Programms durchführen muss, um zu entschlüsseln, was ein Methodenaufruf bewirken könnte. Genau dies soll der Begriff der Substituierbarkeit bringen.

54.1 Der Begriff der Substituierbarkeit

Zuweisungskompatibilität zwischen verschiedenen Typen bedeutet, dass Objekte eines Typs da auftreten dürfen, wo Objekte eines anderen Typs erwartet werden. Wenn das gutgeht, also wenn durch eine entsprechende Zuweisungskompatibilität keine Fehler entstehen, spricht man von der **Substituierbarkeit** der Objekte des Typen auf der linken Seite der Zuweisung durch die des Typen auf der rechten.

Nun ist die Frage, ob eine Zuweisung gutgeht, eine, die man gern automatisch, am besten durch den Compiler, beantwortet hätte. Der Begriff der Substituierbarkeit ist daher in der Programmierung zu einem eigenständigen geworden, der zunächst unabhängig von der (an *Typkonformität* gebundenen) Zuweisungskompatibilität betrachtet werden kann. Der Begriff der Substituierbarkeit soll daher zunächst einmal genauer untersucht werden.



strenge Auslegung des Begriffs der Substituierbarkeit

In der strengsten Auslegung des Begriffs der Substituierbarkeit kann ein Objekt ein anderes nur dann substituieren, wenn sich das auf den Programmablauf in keiner Weise auswirkt. Dazu müsste das ersetzende Objekt aber nicht nur gleich implementiert sein wie das ersetzte (also Instanz derselben Klasse⁸⁵ sein), sondern sich auch noch (zum Zeitpunkt der Substituierung) im selben Zustand wie das substituierte befinden. Wenn nämlich beispielsweise eine Instanz der Klasse **Stack** gerade leer ist, ist sie nicht grundsätzlich gegen eine, die gerade nicht leer ist, austauschbar: Eine Operation **pop**, die das oberste Element des Stacks liefern soll, würde im einen Fall scheitern, im anderen Fall nicht. Da sich Objekte aber nicht abnutzen (so dass sie aus Wartungsgründen substituiert werden müssten), gibt es wohl kaum einen Grund für eine Substituierung sich identisch verhaltender Objekte und damit auch nicht für einen entsprechend eng gefassten Substituierbarkeitsbegriff.

Der Substituierbarkeitsbegriff muss also zumindest vom konkreten Zustand der Objekte unabhängig sein. Das hat den Vorteil, dass man die Betrachtung von Substituierbarkeit von der Laufzeit auf die Übersetzungs- (oder Entwurfs-) Zeit verlagern kann. Auf dieser Ebene ist aber zumindest das Verhalten aller Objekte einer Klasse gleich (nämlich durch dieselbe Klassendefinition) spezifiziert, so dass eine gegenseitige Substituierbarkeit von Objekten derselben Klasse automatisch gegeben ist.

Interessant wird die Frage der Substituierbarkeit erst, wenn die Objekte nicht derselben Klasse angehören und wenn man eine gewisse Abweichung im Verhalten von zu substituierenden Objekten zulässt. So könnte man sich beispielsweise vorstellen, dass ein substituierendes Objekt funktional äquivalent ist (also das Gleiche tut), aber auf eine andere Art. Es könnte z. B. seinen Dienst schneller verrichten als das substituierte oder mit weniger Speicheranforderungen. Diese sog. *nichtfunktionalen Anforderungen*, die normalerweise von den *funktionalen* getrennt dargestellt werden, sind aber in Wirklichkeit gar nicht immer hundertprozentig davon zu trennen und es ist durchaus vorstellbar, dass ein Programm, das von einem funktionierenden nur in nicht-funktionalen Eigenschaften abweicht, nicht funktioniert (beispielsweise weil bestimmte angenommene Echtzeitbedingungen nicht eingehalten werden und dies zu Abbrüchen durch Time outs o. ä. führt).

Substituierbarkeit bei funktional äquivalentem Verhalten

Ein klassisches Beispiel für die gegenseitige Austauschbarkeit funktional äquivalenter, aber verschiedener Typen ist die plattformunabhängige GUI-Programmierung. So basiert beispielsweise die GUI-Programmierung von und mit ECLIPSE auf einer Reihe von Typen, deren Objekte für die Elemente eines GUI stehen, also Fenster, Buttons etc. Nun hat jedes Betriebssystem seine eigenen, den jeweiligen Eigenheiten angepassten Implementierungen dieser GUI-Elemente. Es ist also sinnvoll, für jeden Typ

Beispiel GUI- Programmierung

⁸⁵ Solange Typdefinitionen nur Syntax beschreiben, ist das Verhalten der Objekte durch ihren Typ nicht festgelegt. Selbst bei einer abstrakten Spezifikation des Verhaltens (z. B. durch einen abstrakten Datentyp) wird aber in der Regel nur funktionales Verhalten beschrieben und keine nichtfunktionalen Aspekte wie beispielsweise Zeitverhalten. Auch aus nichtfunktionalen Anforderungen kann sich aber eine mangelnde Substituierbarkeit ergeben (s. u.).



von GUI-Element eine Reihe von Subtypen, einen pro Betriebssystem, anzubieten, die die Elemente auf die jeweiligen Implementierungen des Betriebssystems abbilden. Objekte dieser Typen sind innerhalb derselben Gruppe (also als Objekte von Subtypen desselben Typs) funktional äquivalent, können sich aber in Aussehen und ggf. auch Detailverhalten (gegenüber dem Bediener) unterscheiden. Diese Unterschiede sind jedoch gewollt und die Substituierbarkeit bleibt davon unberührt.

Aber auch damit ist noch nicht Schluss mit der Auslegung des Begriffs von der Substituierbarkeit. Es ist z. B. denkbar, dass unterschiedliches Verhalten nicht nur toleriert, sondern sogar gewünscht wird. Denken Sie beispielsweise an einen Editor, der eine Funktion „rückgängig machen“ hat, die es erlaubt, den Effekt der letzten Aktion, die Sie ausgeführt haben, zurückzunehmen, und zwar unabhängig davon, welche Aktion dies war. Die Aktionen, die möglich sind, sind in der Regel höchst unterschiedlich, so dass es keinen einheitlichen Mechanismus gibt, der erlauben würde, jeden Effekt auf die gleiche Weise rückgängig zu machen. Es ist also sinnvoll, Aktionen als Objekte zu repräsentieren, die neben einer Funktion „ausführen“ auch noch eine „Rückgängigmachen“ haben, die, für jeden Typ von Aktion verschieden, das jeweils Notwendige verrichtet. Objekte all dieser Aktionstypen wären dann, was den Tatbestand der Ausführ- und Rückgängigmachbarkeit angeht, gegeneinander austauschbar und die Aktionstypen wären alle Subtypen eines allgemeinen (abstrakten) Typen **Aktion**, obwohl ihr konkretes Verhalten, also das, was jeweils mit „ausführen“ und „rückgängig machen“ verbunden ist, jeweils höchst unterschiedlich ausfällt. Die Anforderungen an die Substituierbarkeit sind in diesem Fall also eher gering.

Substituierbarkeit bei funktional nicht äquivalentem Verhalten

Im allgemeinen als nicht gegeneinander austauschbar angesehen wird jedoch Verhalten, bei dem eine Funktion, die in dem auszutauschenden Typen definiert ist, in dem austauschenden Typ schlicht fehlt. So wäre beispielsweise eine Aktion, für die „rückgängig machen“ nicht definiert ist (z. B. **Speichern**), kein Subtyp von **Aktion**, da Objekte dieses Typs nicht überall da auftauchen können, wo allgemein Aktionen erwartet werden. Es ist diskutierbar, ob es ausreicht, die Funktion „rückgängig machen“ in **Speichern** leer zu implementieren, also beispielsweise nichts passieren zu lassen oder eine Meldung „rückgängig Machen leider nicht möglich“ auszugeben; die Benutzerin ist vermutlich zerknittert, aber das Programm würde immerhin weiterlaufen. Die formale Spezifikation, nämlich die Aktion (das Speichern) rückgängig zu machen, würde freilich nicht erfüllt; Objekte vom Typ **Speichern** sind damit strenggenommen nicht gegen andere Objekte vom Typ **Aktion** austauschbar.

hinreichende Ausschlusskriterien für die Substituierbarkeit

54.2 Subtyping und das Prinzip der Substituierbarkeit

Das Beispiel von **Aktion** und **Speichern** legte bereits nahe, dass die Substituierbarkeit immer dann fraglich ist, wenn keine Typerweiterung vorliegt, wenn man es sogar insbesondere mit einer Typeinschränkung zu tun hat. Dies soll nun etwas genauer beleuchtet werden.



In JAVA ist die Klasse **Stack** als Subklasse der Klasse **Vector** (die keinen Vektor im mathematischen Sinne, sondern eher ein dynamisches, also in seiner Größe wachsen könnenndes Array repräsentiert) definiert. Das führt jedoch dazu, dass an Stellen im Programm, an denen eine indizierte Sammlung von Objekten mit wahlfreiem Zugriff (eben ein Objekt vom Typ **Vector**) erwartet wird, ein Objekt vom Typ **Stack** auftauchen kann, dessen interne Repräsentation zwar auf einem dynamischen Array á la **Vector** aufbauen mag (und der deswegen davon erbt), der aber an seiner öffentlich zugängigen Schnittstelle die Funktionen für den wahlfreien Zugriff unterdrücken muss (was in JAVA allerdings nicht geht). Sollte nämlich das Programm, in Erwartung einer indizierten Sammlung, auf ein Element darin zugreifen wollen und anstelle dieser einen Stack vorfinden, kann das Programm nicht fortgesetzt werden, es sei denn, es findet vor dem Zugriff eine entsprechende Prüfung (und ggf. eine Verzweigung zu alternativen Verfahrensweisen) statt. Eine solche Prüfung muss jedoch zur Laufzeit stattfinden; wird sie vergessen (nicht implementiert) und es taucht an dieser Stelle ein Stack auf, dann hat man es mit einem waschechten Programmierfehler zu tun. Immerhin lassen sich solche Fehler einfach vermeiden, indem man abgeleiteten Typen verbietet, Eigenschaften zu unterdrücken.

Stack als Subklasse von **Vector**?

Eine schwächere Variante, die aber ähnlich katastrophale Folgen haben kann und deren Vorliegen nur schwer festzustellen ist, stellt der Fall dar, dass eine Funktion in einem Subtyp so abgeändert wird, dass sie dem (vom Supertypen) erwarteten Verhalten widerspricht. Dies ist beispielsweise bei den beiden Typen **Set** (Menge) und **Bag** (Multimenge) der Fall. Wenn man nämlich **Set** als Subtyp von **Bag** annimmt, so könnte man das durchaus als eine Typeinschränkung begreifen, und zwar eine, in der die Anzahl der Vorkommen jedes einzelnen Elements auf die Werte 0 und 1 (den Wertebereich $\{0, 1\}$) beschränkt ist. Die Funktionen „Hinzufügen eines Elements“, „Entfernen eines Elements“ sowie die Angabe der Größe und der Test auf Enthaltein eines Elements werden von **Set** genau wie von **Bag** unterstützt; der einzige Unterschied scheint zu sein, dass das Hinzufügen eines Elements, das in der Menge schon enthalten ist, diese nicht verändert.

nichtkonforme Verhaltensänderung

Bei genauerem Hinsehen ergibt sich aber das Problem, dass **Set** durch seine Eigenheit mehrere charakteristische Eigenschaften von Multimengen verletzt. So gilt für Sets beispielsweise nicht wie für Bags, dass jedes Hinzufügen eines Elements die Größe um 1 anwachsen lässt. Auch gilt nicht, dass genauso viele Elemente entnommen werden können, wie hinzugefügt wurden; alle doppelten Einfügungen werden von **Set** einfach unterschlagen. Ein Programm, das auf die Eigenschaften von Bags setzt und stattdessen mit einem Set arbeiten muss, funktioniert mit hoher Wahrscheinlichkeit nicht mehr korrekt.

Umgekehrt würde, wenn man **Bag** als Subtyp von **Set** annehmen würde, die für **Set** charakteristische Eigenschaft, nämlich dass nach dem Entfernen eines Elements dieses nicht mehr darin enthalten ist, verletzt. Programme, die darauf bauen, dass diese Eigenschaft garantiert wird und die anstelle einer Menge eine Multimenge bekommen, funktionieren nicht mehr korrekt. Je nach eingenommenem Standpunkt stellt dies einen Typfehler dar.



Wenn man versucht, der Ursache des Fehlers auf den Grund zu gehen, kommt man schnell zu der Einsicht, dass die charakteristischen Eigenschaften keiner der beiden Typen die des jeweils anderen implizieren, dass sie genauer im Widerspruch zueinander stehen. Deswegen lassen sich keine korrekten Sätze wie „für alle Objekte vom Typ X gilt, …“ bilden, wobei die Objekte vom Typ X (**Bag** oder **Set**) immer auch die vom jeweils anderen Typen Y, der Subtyp von X sein soll, einschließen soll. Dies entspricht jedoch genau der Definition des Subtypings aus Kurseinheit 3, Abschnitt 26.1; tatsächlich ist es mehr oder weniger eine Frage der Auslegung, ob für den Tatbestand des Subtypings die in Kapitel 26 genannten syntaktischen Bedingungen, insbesondere die Ko- und Kontravarianz, ausreichen oder ob strengere Bedingungen der Substituierbarkeit eingehalten werden müssen.

In der Praxis wird die Prüfung der Substituierbarkeit durch Type-checking-Verfahren in Form der Prüfung der Typkonformität immer nur angenähert; tatsächlich kann nicht einmal eine Substituierbarkeit ausgeschlossen werden, wenn mangelnde Typkonformität vorliegt.⁸⁶ Gleichwohl werden entsprechende Zuweisungen nicht zugelassen. Umgekehrt bedeutet aber Typkonformität nicht automatisch auch Substituierbarkeit — dazu ist auch eine Betrachtung des Verhaltens notwendig.

**keine umfassende
Prüfung der
Substituierbarkeit in
der Praxis**

54.3 Verhaltensbasiertes Subtyping

Die Regeln des Subtyping aus Kapitel 26 und die damit verbundene Regelung der Zuweisungskompatibilität bezogen sich ja lediglich auf die Elemente einer Typdeklaration und damit auf rein statische Information. Um nun auch das Verhalten der Objekte eines Typs einzufangen, greift man auf eine Idee der formalen Programmverifikation zurück: der Überführung der Vorbedingungen in Nachbedingungen nach der Art von (54.1). Ins Objektorientierte übertragen heißt das, dass ein Typ dann korrekt (implementiert) ist, wenn für jede Methode gezeigt werden kann, dass aus der Vorbedingung der Methode die Nachbedingung folgt (und dass die Invarianten des Typs höchstens temporär, während der Methodenausführung verletzt werden). Wir schreiben dazu für eine Methode m und einen Typ T

$$pre_m^T(self:T) \Rightarrow post_m^T(self:T) \quad (54.2)$$

und meinen damit, dass für eine Implementierung von m in der zu T gehörenden Klasse, die auf einem Empfängerobjekt vom Typ T (einer Instanz der entsprechenden Klasse) aufgerufen wird, die Nachbedingung aus der Vorbedingung folgt. Diesen Beweis müssen wir aber zum Glück nicht führen — wir sind hier nicht an der Korrektheit von Implementierungen an sich interessiert, sondern vielmehr daran, ob sich eine (korrekte) Implementierung durch eine andere (ebenfalls korrekte, aber eben andere, auch in Bezug auf ihre Spezifikation)

⁸⁶ Die Substituierbarkeit kann deswegen nicht vollständig ausgeschlossen werden, weil diese immer auch vom Verwendungskontext abhängig ist — wenn beispielsweise mit einem Objekt gar nichts gemacht wird, kann es auch durch ein anderes ersetzt werden, selbst wenn die entsprechenden Typen nicht konform sind. Mehr dazu in Abschnitt 54.5.



ersetzen lässt. Konkret: Wir sind an einer verhaltensbasierten Subtypenrelation interessiert, also an den Bedingungen, die potentielle Subtypen einhalten müssen, damit sie die Spezifikation des Supertyps erfüllen, so dass man sie als verhaltenskonform betrachten kann und eine Subtypenbeziehung wie in Kapitel 26 beschrieben gegeben ist. Das ist immer dann der Fall, wenn obige Implikation auch für Objekte des potentiellen Subtypen S gilt, also wenn

$$pre_m^T(self:S) \Rightarrow post_m^T(self:S) \quad (54.3)$$

Man spricht dann von einem **Behavioural subtyping**, das zu deutsch am besten als **verhaltensbasiertes Subtyping**⁸⁷ wiedergegeben wird.

Es gilt also, (6.3) sicherzustellen. Bei der Spezifikation der Methoden des (potentiellen Sub-) Typs S wird man aber zunächst nicht auf die Vor- und Nachbedingungen von T zurückgreifen, sondern eigene angeben, so dass für alle Methoden m von S

$$pre_m^S(self:S) \Rightarrow post_m^S(self:S) \quad (54.4)$$

als Ausdruck der Korrektheit gilt. (54.3) folgt daraus unmittelbar, wenn

$$pre_m^S \equiv pre_m^T \text{ und } post_m^S \equiv post_m^T$$

für alle m in T ist, aber das wird ja wie gesagt im allgemeinen nicht der Fall sein. Die Frage ist vielmehr: Wie müssen pre_m^S , pre_m^T , $post_m^S$ und $post_m^T$ miteinander im Verhältnis stehen, damit Objekte vom Typ S die Anforderungen für Objekte vom Typ T erfüllen? Formal: Was müssen wir voraussetzen, damit aus (54.4), dem Verhalten von m in S , (54.3), das Verhalten von m in T angewandt auf Objekte aus S , folgt?

Leider ist es mit der Beantwortung dieser Frage aber noch nicht genug. Aufgrund des in der objektorientierten Programmierung weit verbreiteten Aliasing kann ein Objekt vom Typ S , das von einem Klienten wie ein Objekt vom Typ T betrachtet wird, von einem weiteren Klienten wie ein Objekt vom Typ S (oder wie von einem anderen Supertypen als T) betrachtet werden. Dadurch können dann auch Methoden auf dem Objekt aufgerufen werden, die Zustandsänderungen des Objekts verursachen, die nicht durch die mit T verbundenen Methodenspezifikationen (deren Vor- und Nachbedingungen) abgedeckt sind, ja die ein Verhalten bewirken, das mit dem von T nicht kompatibel und das für Benutzerinnen des Objekts, die es als ein T ansehen, nicht akzeptabel ist. Eine methodenweise Betrachtung von Bedingungen für die Substituierbarkeit reicht also nicht aus. Man ahnt bereits, dass die Angelegenheit komplex wird.

zusätzliches Problem
Aliasing mit anderen
Typen

⁸⁷ Nun ist das immer noch halb englisch — „das Subtypen“ klingt aber wirklich doof.



54.4 Das Liskov-Substitutionsprinzip

In Sachen verhaltensbasiertes Subtyping am meisten Bekanntheit erlangt haben die Arbeiten von Barbara Liskov und Jeannette Wing. Tatsächlich ist das sog. **Liskov-Substitutionsprinzip** (Liskov substitution principle, LSP) eines der am häufigsten zum Thema Subtyping angeführten, weswegen es auch hier behandelt werden soll. Ohne den Beitrag der beiden schmälern zu wollen, ist dies durch die Sache jedoch nicht gerechtfertigt — anderen, früheren Arbeiten gebührt mindestens gleicher Ruhm und außerdem ist, wie Sie noch sehen werden, das LSP zu streng gefasst, weswegen es nützliche, für die Praxis relevante Fälle des verhaltensbasierten Subtyping ausschließt.



Historischer Hintergrund des Liskov-Substitutionsprinzips war die Suche nach einer hinreichenden Bedingung für die Subtypenrelation zwischen zwei Typen. Wir hatten ja in Kapitel 26 (Kurseinheit 3) festgestellt, dass es bei den meisten Programmiersprachen ausreicht, dass ein Typ B deklariert, Subtyp eines Typs A zu sein, damit Zuweisungskompatibilität von B nach A festgestellt werden kann.⁸⁸ Dazu war es allerdings notwendig, dass die Eigenschaften von A auf B übertragen (vererbt) und dass dabei die Regeln von den ko- bzw. kontravarianten Redefinitionen von Parametertypen eingehalten werden. Dies wird im allgemeinen durch die Sprachdefinition und durch den Compiler sichergestellt.

Motivation

Das *verhaltensbezogene Subtyping* geht nun über die auf die Kontrolle der Parametertypen beschränkte, syntaktische Subtypenbeziehung hinaus, indem es — nach Liskov und Wing — fordert, dass sich Objekte eines Subtyps und seines Supertyps gleich verhalten sollen, und zwar insoweit irgend jemand oder irgendein Programm dies feststellen kann. Diese Forderung kulminiert in der Regel

verhaltensbezogenes Subtyping

Subtype Requirement: Let $\phi(x)$ be a property provable about objects x of type T . Then $\phi(y)$ should be true for objects y of type S where S is a subtype of T .



Eine deutsche Paraphrase dessen fanden Sie bereits in Abschnitt 26.1.

kritische Betrachtung

Diese Definition ist gleich in mehrfacher Hinsicht problematisch.

1. Da sie die Subtypenbeziehung definiert, ist davon auszugehen, dass x exakt vom Typ T ist und y exakt vom Typ S . Es sind also insbesondere x und y keine Objekte von Subtypen von T bzw. S . Damit ist die Definition nicht auf abstrakte Typen und Interfaces ausdehnbar.
2. Damit zusammenhängend ist die Aussage losgelöst von jedem konkreten Gebrauch der Objekte. Wie in Abschnitt 54.5 noch genauer dargestellt werden wird, kann

⁸⁸ Man beachte den Zirkel in der Definition: Für das Subtyping reicht aus, dass Zuweisungskompatibilität besteht; der Compiler leitet die Zuweisungskompatibilität aus dem Bestehen einer Subtypenbeziehung (z. B. `extends` oder `implements` in JAVA) ab.



eine Substituierbarkeit in einem gegebenen Kontext sehr wohl bestehen, auch wenn die Typen nach obiger Definition nicht substituierbar sind. Man würde die Anforderungen in einem solchen Fall in einem Interfacetypen festhalten, der nur die im Kontext benötigten Eigenschaften spezifiziert. Dieser Typ hat dann aber (gemäß Punkt 1) keine Objekte x.

3. Die Aussage ist implizit allquantifiziert über ϕ , d. h., sie soll für alle möglichen Eigenschaften (Prädikate) ϕ gelten. Das bedeutet wiederum, dass alle Eigenschaften von T auch für S gelten — die Objekte von S müssen sich also, sieht man einmal von zusätzlichem Verhalten ab, exakt gleich verhalten. Das aber stellt die Idee des Subtyping weitgehend infrage: Wenn ein Objekt sich von dem, das es ersetzen soll, überhaupt nicht unterscheidet, wozu brauche ich es denn dann überhaupt?

Zu Punkt 3 ist abschwächend zu sagen, dass hier vermutlich Verhalten des Programms insofern unverändert sein soll, als es immer noch seinen Zweck erfüllt, d. h., seiner Spezifikation genügt. Das ist jedoch etwas anderes, als in dem Prinzip ausgedrückt wird.

Eine Subtypenrelation zwischen S und T, die das obige Subtype requirement erfüllt, definieren Liskov und Wing zunächst wie folgt:

Subtypenrelation nach Liskov und Wing

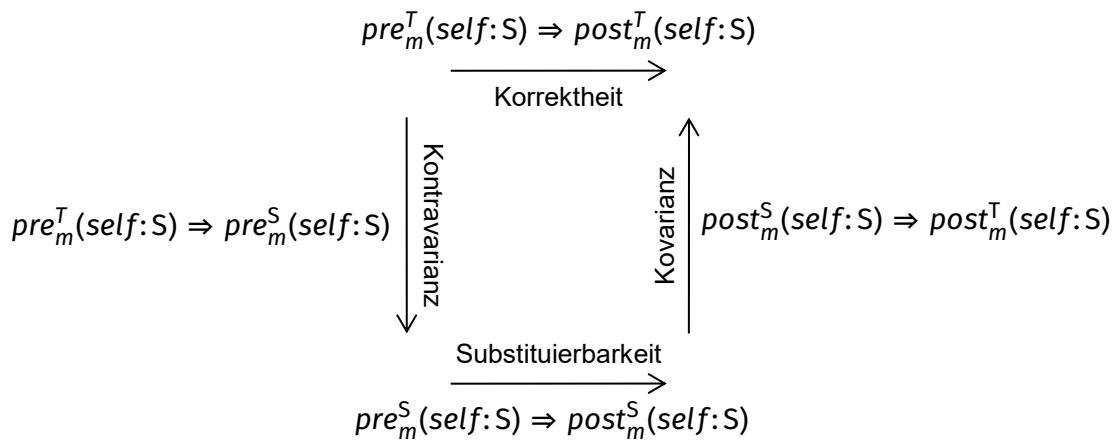
1. Überschreibende Methoden in S erhalten das Verhalten der überschriebenen Methoden in T. Dazu gehört:
 - a. Kontravarianz der Argumenttypen der überschreibenden Methode
 - b. Kovarianz des Ergebnistyps der überschreibenden Methode
 - c. Kovarianz der Ausnahmen der überschreibenden Methode (die Typen der geworfenen Exceptions sind entweder Subtypen von den Typen der Exceptions der überschriebenen Methode oder die Exceptions werden gar nicht geworfen; keinesfalls kommen Exceptions hinzu)
 - d. Vorbedingungen der überschriebenen Methode implizieren Vorbedingungen der überschreibenden: $pre_m^T(self:S) \Rightarrow pre_m^S(self:S)$
 - e. Nachbedingungen der überschriebenen Methode werden von Nachbedingungen der überschreibenden Methode impliziert, also gilt hier: $post_m^S(self:S) \Rightarrow post_m^T(self:S)$
2. Die Invarianten von S implizieren die von T.

Man beachte, dass die (auch semantisch genannten) Regeln 1.d und 1.e die (auch syntaktisch genannten) Regeln 1.a – 1.c implizieren: In typlosen Sprachen wie SMALLTALK beispielsweise würde man die entsprechenden Anforderungen an die Parameterobjekte mittels Vor- und Nachbedingungen formulieren und die Menge der geworfenen Exceptions wäre Teil der Nachbedingungen.

Man kann sich die Wirksamkeit der Bedingungen wie folgt erklären:

Veranschaulichung





Aus den obigen Regeln der Subtyprelation und aus der Korrektheit der überschreibenden Methode in S folgt die gewünschte Substituierbarkeit.

Es bleibt jedoch noch das eingangs beschriebene, mit dem Aliasing und der damit verbundenen Möglichkeit des zusätzlichen Methodenaufrufs assoziierte Problem bestehen. Dafür identifizieren Liskov und Wing zwei Lösungen:

Die erste sagt aus, dass das Verhalten eines Subtyps immer dann mit dem des Supertyps konform ist, wenn alles zusätzliche Verhalten des Subtyps durch Verhalten des Supertyps simuliert werden kann, wenn es also für jede zusätzliche Methode eine Kombination von Methodenaufrufen des Supertyps (bzw. deren überschriebenen, verhaltenskonformen Varianten) gibt, die den Effekt der zusätzlichen Methode hat und die der Klient des Objekts auch selbst (oder ein anderer Klient, der aber das Objekt durch denselben Typ sieht) durchführen könnte. Diese Bedingung ist jedoch ziemlich hart, da sie im Grunde aussagt, dass ein Subtyp lediglich Makros hinzufügen darf. Auf der anderen Seite ist ihre Einhaltung, wenn auch nicht automatisch, so jedoch zumindest anschaulich relativ einfach nachzuweisen (was für die Praxis eminent wichtig ist).

Die zweite Lösung ist denn auch weniger einschränkend, dafür aber in der Praxis kaum nachzuweisen: Sie verlangt von jedem Typ zusätzlich zur Verhaltensspezifikation der Methoden (über die üblichen Vor- und Nachbedingungen) die Einhaltung von Verlaufseigenschaften, die die möglichen Zustandswechsel der Objekte des Typs betreffen und die wir deswegen **Zustandswechselinvarianten** nennen. Dazu wäre eine Art endlicher Automatenspezifikation, also eine Spezifikation der Menge der möglichen Zustände und Zustandsübergänge, notwendig. Das Problem dabei ist jedoch, dass die Zustände eines Objekts nicht abstrakt (z. B. in Form einer Aufzählung von Zustandsnamen) existieren, sondern sich in der Belegung von Instanzvariablen manifestieren, so dass sich der Zustandsraum kombinatorisch (abzüglich der funktionalen Abhängigkeiten der Attribute) ergibt. Um dem aus dem Weg zu gehen, werden bei den historischen Zustandswechseln lediglich zwei beliebige, zeitlich nicht notwendig unmittelbar aufeinander folgende Zustände betrachtet und für diese eine Bedingung formuliert, die eingehalten werden muss. Man beachte, dass die Einhaltung der ersten Bedingung die zweite impliziert: Wenn alle Methoden des Supertyps die historischen Invarianten einhalten und wenn eine Methode

das Aliasing-Problem

Verhaltenssimulation

Zustandswechselinvarianten



des Subtyps sich als Kombination von Methoden des Supertyps darstellen lässt, dann hält auch diese die historischen Invarianten ein.

So gut das Liskov-Substitutionsprinzip auch begründet sein mag, es erweist sich für die Praxis als zu streng. So könnte man beispielsweise in einem Typ **T** eine Methode `echo : i <Integer> ^ Integer` definieren, die den Eingabeparameter gleich wieder ausgibt (die sog. Identität oder Identitätsfunktion). In einem Subtyp **S** könnte man dann die Methode so überschreiben, dass sie beliebige Objekte entgegennimmt und gleich wieder zurückgibt: `echo : i <Object> ^ Object`. Die Methode ist damit zwar in ihrem Eingabeparameter kontravariant, aber in ihrem Ausgabeparameter nicht kovariant, und erfüllt somit die Bedingung des LSP nicht. Gleichwohl kann `echo :` auf einem Objekt vom Typ **S** überall da aufgerufen werden, wo es auch auf einem Objekt von Typ **T** aufgerufen werden kann, denn es kommt so niemals dazu, dass die Regel der Kovarianz des Ausgabeparameters verletzt wird. Wir haben es beim LSP also mit einer Typprüfung zu tun, die gültige Programme zurückweist. Dafür, dass die Einhaltung des LSP für den allgemeinen Fall gar nicht automatisch überprüft werden kann, ein ziemlich hoher Preis.

LSP ist zu restriktiv

54.5 Relativität der Substituierbarkeit

Nun kann man argumentieren, dass Substituierbarkeit im Einzelfall von der jeweiligen Verwendung der Typen abhängt (vgl. Fußnote 86). So kann man im Beispiel von **Set** und **Bag** gar nicht sagen, dass grundsätzlich das eine nicht das andere substituieren kann, denn es kann durchaus Verwendungen des einen oder anderen Typen geben, in denen der jeweils andere durchaus als Ersatz infrage kommt. Das Vorliegen einer echten Substituierbarkeit ist also nicht auf Basis der beteiligten Typen allein entscheidbar, sondern hängt auch von deren Verwendung ab. Das wiederum bedeutet (auch vor dem Hintergrund des oben gesagten zur funktionalen Äquivalenz), dass es so etwas wie eine absolute (d. h., von jeder möglichen Verwendung losgelöste) Substituierbarkeit eigentlich nicht gibt.

Wie kann man diesem Umstand abhelfen? Nun, indem man neben der Sicht der Programmiererin, die die Typen liefert und die sich um deren Substituierbarkeit (und die davon abhängige Subtypenbeziehung) sorgt, auch noch die Sicht der Programmiererin, die die Typen zu einem konkreten Zweck einsetzen will, einbezieht. Nur diese Programmiererin weiß nämlich, was sie sich genau von einem Typen (genauer: von den Objekten eines Typs) erwartet, und nur diese kann beurteilen, welche Typen zu ihrem Zweck gegeneinander austauschbar sind.

Perspektivwechsel

Wie aber drückt die Programmiererin ihre Erwartung aus? Die Antwort ist verblüffend einfach: Indem sie selbst einen Typ definiert, dessen Definition die — und eben nur die — Eigenschaften umfasst, die sie verlangt. Ein Typ, der genau diese Erwartung und nicht mehr ausdrückt, spezifiziert gewissermaßen eine *Rolle*, die die Objekte im Kontext der Verwendung zu spielen haben. In der Regel wird dieser Typ nur einen Teil der Eigenschaften umfassen, die allgemeine Typen wie **Set** oder **Bag** spezifizieren; im Gegenzug ist damit aber auch die Auswahl der Typen, deren Objekte für ihre Zwecke infrage kommen, größer.

Typen als Rollen



Die einzige Voraussetzung dafür, dass Objekte die ihnen durch ihren Typ aus Klientensicht zugewiesene Rolle auch spielen können, ist, dass sie dazu typkonform sind.

Diese zweiseitige Sicht auf Typen, nämlich die der Nutzerin und die der Anbieterin, beginnt sich erst langsam durchzusetzen. Die Programmiersprache JAVA und in der Folge auch C# bieten immerhin ein *Interface-als-Typ-Konzept* (s. Kapitel 40 in Kurseinheit 4 und Abschnitt 50.4.2 in Kurseinheit 5) an, mit dem es möglich ist, in einem Programm partielle Sichten auf Typen zu spezifizieren. Allerdings wird in der Programmierpraxis die Möglichkeit dieser beiden Sprachen kaum dazu genutzt, Benutzerinnen ihre eigenen Anforderungen als Typen definieren zu lassen. Vielleicht handhaben Sie es ja zukünftig anders.

55 Das Fragile-base-class-Problem

Unter dem **Fragile-base-class-Problem** versteht man eine ganze Familie von Problemen, die in unmittelbarem Zusammenhang mit der Vererbung stehen. Dabei ist der Name insoweit etwas irreführend, als nicht unbedingt die *Basisklassen*, also die Superklassen (vgl. Abschnitt 11.1), die „anfälligen“ oder „zerbrechlichen“ sind, sondern eher die Klassen, die von ihnen erben. Ein einfaches Beispiel soll erläutern, worum es geht.

Wir nehmen an, wir hätten eine Klasse **TapeArchive** geschrieben, die Videobänder archiviert. Da es sich nicht um eine Mickey-Maus-Anwendung auf dem heimischen Desktop handelt, sondern um ein kommerzielles System, erfolgt die Datenhaltung in einer Datenbank. Die folgende SMALLTALK-Klassendefinition gibt einen kleinen Ausschnitt des Systems wieder.

Klasse	TapeArchive
benannte Instanzvariablen	database
Klassenmethoden	
1510 new	
1511 ^ super new initialize	
Instanzmethoden	
1512 initialize	
1513 database := Database new: 'tape archive'	
1514 addTape: aTape	
1515 database beginTransaction.	
1516 database add: aTape.	
1517 database endTransaction	
1518 addAllTapes: aCollection	
1519 aCollection do: [:aTape self addTape: aTape]	



Nun ist es in der objektorientierten Programmierung üblich, dass man solche Klassen wiederverwendet, indem man davon neue Klassen ableitet. Wenn beispielsweise eine Kundinenanforderung kommt, nach der mit jeder Archivierung eine Nachricht verschickt werden



muss, die das anzeigt, und wenn diese neue Anforderung nicht für alle Kundinnen der Software gleichermaßen interessant ist, dann schreibt man eine Subklasse, die das geänderte Verhalten bereitstellt:

Klasse	NotifyingTapeArchive
benannte Instanzvariablen	listener
Superklasse	TapeArchive
Klassenmethoden	
1520 new: aListener	
1521 ^ super new initialize: aListener	
Instanzmethoden	
1522 initialize: aListener	
1523 listener := aListener	
1524 addTape: aTape	
1525 super addTape: aTape.	
1526 listener notify: aTape	

Das Schöne an der Objektorientierung ist dabei, dass man nur die Unterschiede (Differentia) spezifizieren muss — der Rest wird einfach geerbt.

offene Rekursion zum Dritten

So genügt es im gegebenen Fall, die Initialisierung (nicht so interessant) und die Methode **addTape:** anzupassen. Das Verhalten von **addAllTapes:** kann unverändert bleiben, da dies die (dynamisch gebundene) Methode **addTape:** aufruft und somit auch das von **NotifyingTapeArchive** geforderte Verhalten, nämlich die Notifikation aller archivierten Bänder, garantiert. Ein ganz ähnliches Beispiel (anhand der Klasse **Collection**) war Ihnen in Kurseinheit 2, Abschnitt 10.3 schon einmal begegnet; es handelt sich auch hier um einen Fall von *offener Rekursion*, die erst durch *dynamisches Binden* (Kapitel 12) aufgelöst wird. Aufrufe dieser Art sind das Herz vieler objektorientierter (Anwendungs-)Frameworks und auch diverser Entwurfsmuster.

Oder auch nicht. Das Problem ist nämlich, dass man der Klasse

Vorsicht Falle

TapeArchive weder ansieht, was sie garantiert, noch, wovon die Korrektheit ihrer Subklassen abhängt. Wenn in der Folge z. B. eine anderere Kundin nörgelt, das Hinzufügen von großen Mengen von Bändern dauere zu lange, wenn man das wiederholte Aufrufen von **addTape:** aus **addAllTapes:** und die dadurch bedingte wiederholte Ausführung von **beginTransaction** und **endTransaction** als Ursache ausmacht und wenn man dann in Erwägung zieht, nicht nur die nörgelnde Kundin in ihrer Version, sondern alle Kundinnen mit der optimierten Implementierung

Klasse	TapeArchive
Instanzmethoden	
1527 ...	
1528 addAllTapes: aCollection	



```
1529     database beginTransaction.  
1530     aCollection do: [ :aTape | database add: aTape ].  
1531     database endTransaction
```

zu beglücken, was spräche dagegen?

Selbsttestaufgabe 55.1

Bevor Sie weiterlesen, antworten Sie: Was spräche dagegen?

Vermutlich nur die wenigsten unter Ihnen werden sofort sagen können, was dagegen spricht, denn in der Klasse `TapeArchive` scheint nach wie vor alles in bester Ordnung zu sein. Was man nämlich nicht sehen kann, ist, dass die Korrektheit der Methode `addAllTapes`: davon abhängt, dass sie `addTape`: aufruft — zwar nicht für die Klasse `TapeArchive` selbst, dafür aber für ihre Subklasse `NotifyingTapeArchive`. Hier werden jetzt nämlich nur noch für einzeln archivierte Bänder Benachrichtigungen verschickt.

Hand aufs Herz: Hätten Sie den Fehler vorhergesagt? Wenn nicht, dann liegt das vermutlich daran, dass Sie der Täuschung erlegen sind, `addAllTapes`: in `TapeArchive` würde die benachbarte Methode `addTape`: aufrufen, und wenn man nur den Beitrag von `addTape`: in `addAllTapes`: verlagert und dafür `addTape`: nicht mehr aufruft, dann wäre das eine semantikerverlustende Umstrukturierung (ein sog. *Refactoring*; s. Kurs 01853). Tatsächlich befreit aber genau dies die Subklassen der Möglichkeit, eigenes Verhalten an genau dieser Stelle — dem Aufruf von `addTape`: — einzubringen, und wenn eine solche Beraubung im Nachhinein erfolgt, kann sie eben den Code „zerbrechen“.

Das Schlimme an diesem Problem ist, dass man noch nicht einmal genau weiß, wem man die Schuld geben soll — `TapeArchive`, weil es einen Vertrag bricht, den es gar nicht paraphiert hat (oder weil es keine Rücksicht auf Subklassen nimmt, die es gar nicht kennt), oder `NotifyingTapeArchive`, weil es sich grundlos darauf verlässt, dass die geerbten Methoden dauerhaft die eigenen (in diesem Fall das geerbte `addAllTapes`: das eigene `addTape`:) aufrufen? Wenn noch nicht einmal die Schuld feststeht — wie kann man das Problem verhindern?

Schuldfrage

Es gibt zahlreiche Varianten des Fragile-base-class-Problems, die hier nicht alle aufgeführt werden sollen. Zugrunde liegt ihnen immer dasselbe: Zwischen einer Klasse und ihren Subklassen bestehen durch die Vererbung von Eigenschaften starke Abhängigkeiten, die — wenn überhaupt — nur unvollständig dokumentiert sind. Zwar könnte man annehmen, dass von allem, was vererbt wird, eine Abhängigkeit ausgeht, die man bei Änderungen pauschal berücksichtigen muss, aber dies würde die Möglichkeiten, in Superklassen etwas zu ändern, so stark einschränken, dass das ganze Konzept ad absurdum geführt würde. Es bleibt also nicht viel mehr, als beim Einsatz von Vererbung große Vorsicht walten zu lassen oder sie ganz zu verbieten.





Implementation inheritance—the ability of one component to "subclass" or inherit some of its functionality from another component—is a very useful technology for building applications. Implementation inheritance, however, can create many problems in a distributed, evolving object system.

The problem with implementation inheritance is that the "contract" or relationship between components in an implementation hierarchy is not clearly defined; it is implicit and ambiguous. When the parent or child component changes its behavior unexpectedly, the behavior of related components may become undefined. This is not a problem when the implementation hierarchy is under the control of a defined group of programmers who can make updates to all components simultaneously. But it is precisely this ability to control and change a set of related components simultaneously that differentiates an application, even a complex application, from a true distributed object system. So while implementation inheritance can be a very good thing for building applications, it is not appropriate for a system object model that defines an architecture for component software.

*In a system built of components provided by a variety of vendors, it is critical that a given component provider be able to revise, update, and distribute (or redistribute) his product without breaking existing code in the field that is using the previous revision or revisions of his component. **In order to achieve this, it is necessary that the actual interface on the component used by such clients be crystal clear to both parties.** Otherwise, how can the component provider be sure to maintain that interface and thus not break the existing clients?*

Auf die objektorientierte Programmierung übertragen ist das Problem also, dass der Vertrag zwischen den Klassen einer Vererbungshierarchie nicht klar definiert ist. Wenn die Super- oder Subklasse ihr Verhalten unerwartet verändert, kann daraus undefiniertes Verhalten verwandter Klassen resultieren. Tatsächlich war die Vererbung von Implementierung aus der Spezifikation von MICROSOFTs Component Object Model (COM) verbannt; stattdessen setzte man voll auf die Vererbung von Interfaces (was wir als Subtyping bezeichnen würden). Inzwischen (mit dem .NET-Framework) ist diese harte Haltung wieder aufgegeben worden, was wohl auch daran liegt, dass hier Komponenten Klassen sind; es bleiben jedoch die in Kapitel 50 (im Kontext von C#) erwähnten Vorbehalte gegenüber dem dynamischen Binden.

Nun liegt ja zunächst nahe, bei der Vererbung das zu tun, was man bei Abhängigkeiten immer macht: Schnittstellen einzuführen. Im Gegensatz zu der Schnittstelle, die einem Klient einer Klasse angeboten wird (der diese Schnittstelle nutzt, indem er

| **Vererbungsinterface** |



seine Variablen mit dem zur Klasse gehörenden Typ deklariert und somit auf Instanzen der Klasse zugreifen kann), handelt es sich bei der Schnittstelle zwischen einer Klasse und ihren Subklassen jedoch um eine etwas anders geartete: Hier gibt es lediglich zwei Variablen `self` (bzw. `this`) und `super`, die allerdings keine Abhängigkeit von anderen Objekten ausdrücken und die zudem nicht explizit typisiert sind. Die damit verbundene Schnittstelle, also die Menge der Eigenschaften, auf die man über diese Variable zugreifen kann, und wo man diese Eigenschaften dann findet (also wo sie definiert sind), muss man sich selbst zusammensuchen. Mit *Information hiding* und dem Verbergen von *Implementationsgeheimnissen* hat das freilich nichts zu tun. Was man vielmehr bräuchte, wäre ein explizites **Vererbungs-interface**.

Einige erste, zarte Ansätze zur Einführung von expliziten Vererbungsinterfaces hatten Sie bereits gesehen: Die Verwendung des Zugriffsmodifizierers `protected` in JAVA, C# und C++ sowie die explizite Deklaration von Überschreibbarkeit und Überschreibung mittels `virtual` und `override` in C# (und C++). Im obigen Beispiel würde die Deklaration von `addTape`: als nicht überschreibbar (und somit als nicht dynamisch, sondern statisch gebunden) verhindern, dass `NotifyingTapeArchive` diese Methode abändert *und sich zugleich* darauf verlässt, dass die geerbte Methode `addAllTapes`: die überschreibende Version von `addTape`: aufruft (sie könnte sie aber immerhin noch neu einführen, aber diese neue Version würde beim Binden in `addAllTapes`: nicht berücksichtigt). `NotifyingTapeArchive` müsste dann wohl oder übel beide Methoden neu implementieren und könnte bei der Gelegenheit selbst dafür Sorge tragen, dass `addAllTapes`: den Fehler nicht macht. Allerdings würde dadurch auch bei einem direkten Aufruf von `addTape`: von außerhalb auf einer Variable vom Typ `TapeArchive`, die eine Instanz vom Typ `NotifyingTapeArchive` hält, die überschreibende Implementierung unberücksichtigt bleiben (da ja keine dynamische Bindung mehr stattfindet). Im Gegensatz dazu würde die Verwendung von `virtual` bei `addTape`: in `TapeArchive` der Programmiererin einen Hinweis darauf geben, dass die Methode in Subklassen für diese relevante Modifikationen enthalten kann, so dass man Aufrufe dieser Methode nicht einfach, wie im obigen Beispiel geschehen, kürzen kann.

`protected`, `virtual` und `override`

Wenn es um die Sicht- und Zugreifbarkeit von Elementen geht, scheint der Zugriffsmodifizierer `protected` zunächst auf gleicher Ebene mit `public` und `friend` zwischen zwei (nicht über Vererbung in Beziehung stehenden) Klassen zu stehen: die `protected` deklarierten Elemente einer Klasse sind wie bei einem *dedizierten Export* (s. Abschnitt 52.2) auch in ihren Subklassen sicht- und verwendbar. Was allerdings nicht so klar ist, ist, dass überschreibende, als `protected` deklarierte Methoden auch für den Code der Superklasse zugreifbar sind: eine `protected` Methode einer Subklasse kann — über das dynamische Binden auf `self` bzw. `this` — aus der Superklasse heraus aufgerufen werden, ohne dass die Subklasse irgendeinen Hinweis darauf enthält. Anders als bei der Zugreifbarmachung mit `public` oder `friend` bei nicht über Vererbung in Beziehung stehenden Klassen kann die Zugreifbarkeit also in beide Richtungen gehen, und zwar abhängig davon, ob die Methode überschrieben wird oder nicht: Wird sie überschrieben, kann die überschreibende (in der Subklasse) von der Superklasse aus aufgerufen werden und die

Bidirektionalität des Protected-Interfaces



überschriebene (in der Superklasse) von der Subklasse (über `super`) — wird sie nicht überschrieben, kann die Methode der Superklasse aus der Subklasse heraus aufgerufen werden.

Es bleibt also ein höchst verworrender Eindruck. Dies ist um so bedauerlicher, als Bjarne Stroustrup selbst kommentierte:



One of my concerns about protected is exactly that it makes it too easy to use a common base the way one might sloppily have used global data. ... In retrospect, I think that protected is a case where "good arguments" and fashion overcame my better judgement and my rules of thumb for accepting new features.

Bei der Definition von JAVA fand das offenbar kein Gehör. Und so bleiben die Schlüsselwörter `protected`, `virtual` und `override` nicht viel mehr als Zeichen des Bewusstseins, dass es das Fragile-base-class-Problem gibt.

56 Das Problem der schlechten Tracebarkeit

Spätestens mit der Verfügbarkeit sog. Hochsprachen und den gleichzeitig immer größer werdenden Programmen kam die Frage auf, was „gute Programmierung“ ausmacht. Eines der Hauptprobleme schlechter Programmierung war schnell identifiziert: die große Diskrepanz zwischen statischem, linearem Programmtext und dynamischem, stark verzweigendem und sich wiederholendem Programmablauf. Eine gute Programmiererin hatte ihre Programme so zu schreiben, dass Programmtext und Programmablauf einander möglichst ähnlich waren, dass genauer die (statische) Struktur des Programms möglichst viele Rückschlüsse über seinen (dynamischen) Ablauf erlaubte. Man wollte also von den Programmiererinnen Klartext.

Ebenso schnell wie das Problem wurde seine Hauptverursacherin ausgemacht: die Goto-Anweisung. Sie erlaubt Sprünge von beliebigen Stellen eines Programms an beliebige andere Stellen des Programms und durchbricht dabei auf brutale Art und Weise das ungemein nützliche **Lokalitätsprinzip** von Programmen: Dinge, die zusammengehören, stehen im Programmtext beieinander. So, und nur so, ist bei Inspektion des Programmtextes unmittelbar klar, wie man an eine Stelle im Programm gelangt ist und, mindestens ebenso wichtig, wie eine Variable ihren Wert bekommen hat.

**Goto-Anweisung vs.
Lokalitätsprinzip**

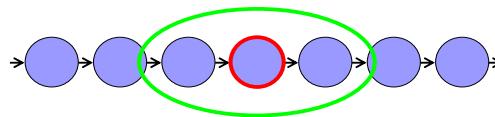
Zur Veranschaulichung soll die nachfolgende Abbildung dienen, die einen Programmtext als eine Folge von Anweisungen stilisiert. Anweisungen sind durch Kreise dargestellt, die (textuelle) Folge der Anweisungen im Programmtext durch die kleinen Pfeile. Ohne besondere, den Kontrollfluss beeinflussende Anweisungen entspricht die (dynamische) Reihenfolge der Ausführung der (statischen) Folge der Anweisungen im Programmtext. Bei

**Anweisungsfolgen
ohne Goto**

Betrachtung des stärker

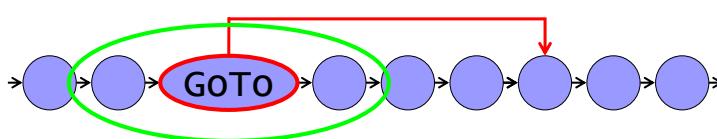


umrandeten, mittleren Kreises (der entsprechenden Anweisung), z. B. während einer Debug-Sitzung, ist daher aus dem unmittelbaren Kontext heraus (der Ellipse; Lokalitätsprinzip!) klar, welche Anweisung davor ausgeführt wurde und welche als nächstes dran kommt. Alles ist in bester Ordnung.



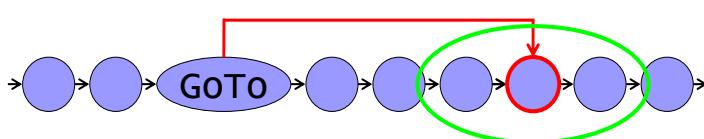
Handelt es sich nun bei einer der Anweisungen um ein Goto, dann ist die Sachlage längst nicht mehr so klar. Man hat vielmehr die folgenden Fälle zu unterscheiden:

1. Die betrachtete Anweisung ist selbst ein Goto:



In diesem Fall ist zwar klar, woher der Programmfluss kommt, und auch, wohin er geht, letzteres aber nur mit einer gewissen Einschränkung — das Ziel ist nicht der Nachbar im Programmtext, sondern befindet sich außerhalb des gewählten Kontextes. Nun kann man den Kontext natürlich so wählen, dass er das Ziel enthält, und kurze Sprünge sind vielleicht auch so innerhalb des betrachteten Kontextes; allgemein gilt aber, dass jede gewählte Lokalität durch einen Sprung verletzt werden kann. Immerhin lässt sich aber das Ziel des Sprungs aus dem Kontext erkennen und der Kontext entsprechend wechseln.

2. Die betrachtete Anweisung ist Ziel eines Gotos:



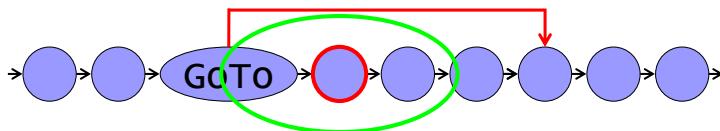
Hier ist das Sachverhalt schon schwieriger. Der Programmfluss scheint bei Betrachtung des Kontextes genau wie im ersten Beispiel zu verlaufen. Wenn man den Kontext allerdings vergrößert, lernt man, dass die dynamische Vorgängerin der betrachteten Anweisung auch ein anderer sein kann. Der Kontext selbst gibt aber keinen Hinweis darauf; zwar kann das Vorhandensein eines Sprunglabels einen Hinweis darauf geben, dass die so markierte Anweisung Ziel eines Gotos sein kann, sie muss es aber nicht; in Sprachen wie BASIC beispielsweise (damals noch weit verbreitet), in denen Zeilennummern gültige Sprungziele sind, muss jede Anweisung als mit einem Label versehen betrachtet werden und kann somit Sprungziel von irgendwoher sein.

Probleme des Goto



Außerdem kann eine Anweisung von verschiedenen Gotos angesprungen werden, so dass unklar bleibt, welches die (zeitliche) Vorgängeranweisung war.⁸⁹

3. Die betrachtete Anweisung ist unmittelbare Nachfolgerin eines Gotos:



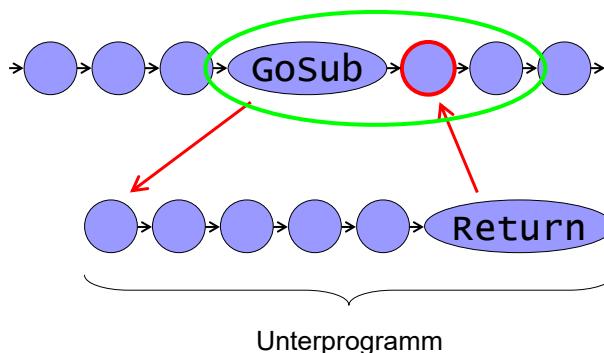
Hier ist zwar aus dem Kontext ersichtlich, dass die statische Vorgängerin nicht die dynamische sein kann, ansonsten kann man aber nur mutmaßen, dass es sich vielleicht um toten Code handeln könnte (also um Code, der niemals ausgeführt wird). Es kann nämlich die Anweisung Sprungziel von Gotos außerhalb des Kontexts (wie in allen anderen Fällen auch) sein.

Fazit: Die Verwendung von Goto-Anweisungen verursacht ein hohes Maß an Nichtwissen bei der Interpretation von Quelltext. Speziell beim Debugging von Programmen ist der Blick in den Quelltext des Programms so nur sehr bedingt von Nutzen. Von daher, so der allgemeine Konsens, ist die Benutzung von Gotos zu vermeiden.

Vermeidung von Gotos

Wenn man also kein Goto benutzen darf, wie steuert man dann den Ablauf von Programmen? Die sog. **strukturierte Programmierung** sieht dafür neben der Sequenz von Anweisungen (ausgedrückt durch die unmittelbare Nachbarschaft im Programmtext) die Verzweigung, die Wiederholung und den Unterprogrammaufruf vor. Von diesen behalten die ersten beiden das Lokalitätsprinzip bei, solange man den Kontext auf den Umfang der *Fallunterscheidung* bzw. Schleife, die damit ausgedrückt wird, ausdehnt. Für den Unterprogrammaufruf gilt das jedoch nicht mehr: Schon weil ein Unterprogramm in der Regel von mehreren Stellen eines Programms aufgerufen werden kann und weil diese Stellen nicht automatisch denselben Kontext haben, wird hier das Lokalitätsprinzip durchbrochen. Dies ist aber unvermeidlich und man tröstet sich damit, dass ein Unterprogramm, genauer eine Prozedur oder eine Funktion, immer genau an die Stelle zurückkehrt, von der es aufgerufen wurde. Es ergibt sich also anschaulich die folgende Situation:

strukturierte Programmierung und Lokalitätsprinzip



⁸⁹ So gesehen gibt es den eingangs geschilderten Fall, bei dem alles klar ist, bei Programmen mit Gotos eigentlich gar nicht.

Bei Betrachtung des textuell unmittelbaren Vorgängers der betrachteten Anweisung sieht man sofort, dass es sich beim dynamischen Vorgänger um die Return-Anweisung des aufgerufenen Unterprogramms handeln muss. Dies ist zwar nicht lokal, aber wenn man sich sicher sein kann, dass das Unterprogramm nur die Variablen manipuliert, die bei seinem Aufruf als tatsächliche Parameter übergeben wurden, und wenn zudem das Unterprogramm bekannte Vor- und Nachbedingungen einhält, dann ist das kein Problem. Selbst wenn man nicht weiß, wie die Variablen manipuliert wurden, so ist die Unwissenheit, die durch einen Unterprogrammaufruf verursacht wird, im Vergleich zu der beim Goto gering. Ihr steht auf der anderen Seite ein großer Nutzen gegenüber:

1. Man vermeidet die Duplizierung von Code, die nötig wäre, wenn man die Anweisungen des Unterprogramms im Aufrufkontext halten wollte und es mehrere solche Aufrufkontakte gibt (das sog. *Inlining*, das manche Compiler aus Optimierungsgründen durchführen).

Man erlaubt der Programmiererin, ihre Programme in Abschnitte zu unterteilen, die sie getrennt untersuchen und verstehen kann.

Besonders der zweite Punkt ist wichtig: Aus Sicht der Programmiererin sollte es nämlich reichen, zu wissen, was ein Unterprogramm tut, um es korrekt benutzen zu können. Sie muss also insbesondere nicht in das Unterprogramm hineinschauen, also seine Anweisungen inspizieren, wenn ihr eigentliches Interesse dem Kontext der Aufrufstelle gilt. Umgekehrt muss sie, wenn sie das Unterprogramm interessiert, nicht wissen, von wo es überall aufgerufen wird — es reicht dann, zu wissen, mit welchen Parametern es versorgt wird, und die sind ihr per formale Parameterdeklaration bekannt. (Voraussetzung dieser Argumentation ist jedoch, dass es keine globalen Variablen gibt, die eine gegenseitige Beeinflussung von Aufrufstelle und Unterprogramm an den tatsächlichen und formalen Parametern vorbei erlauben. Diese globalen Variablen sind jedoch mindestens so sehr verpönt wie das Goto.)



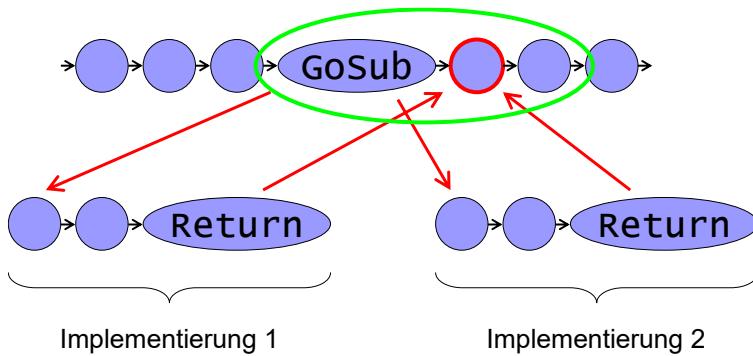
Bei der objektorientierten Programmierung hat man es zunächst mit einer leicht veränderten Situation zu tun. Hier sind nämlich nicht allein das Vermeiden von doppeltem Code sowie die stufenweise Verfeinerung Kriterien für die Aufteilung in Unterprogramme, sondern auch die Disziplin, jede Teilfunktion der Klasse zuzuordnen, deren Daten sie manipuliert. Typische objektorientierte Programme teilen daher die Implementierung größerer Funktionen nicht nur in kleinere auf, sondern verteilen diese auch noch über viele Klassen. Auch wenn es sich dabei



stets nur um Unterprogrammaufrufe handelt, die allen obengenannten Anforderungen genügen, so erfolgen die zum Programmverstehen notwendigen Kontextwechsel doch in so kurzer Folge, dass man schnell den Überblick darüber verliert.⁹⁰

Nun ergibt sich aber mit der Einführung von dynamisch gebundenen Unterprogrammaufrufen, wie sie ja für die objektorientierte Programmierung prägend sind, das Problem, dass aus dem Programmtext nicht unmittelbar ersichtlich ist, wohin der Sprung geht: Wie bereits in Kurseinheit 1, Abschnitt 4.3.2 bemerkt, verbindet das dynamische Binden den Unterprogrammaufruf mit der Verzweigung. Anschaulich betrachtet findet man im Quelltext die folgende Situation vor:

dynamisches Binden als Störenfried



Es ist an der Stelle der betrachteten Anweisung nicht klar, von woher der in der Anweisung zuvor angestoßene Unterprogrammaufruf zurückkehrt — es könnte von jeder Implementierung der im Gosub genannten Methode sein. Um das Sprungziel und damit die Return-Anweisung, die unmittelbarer Vorgänger war, zu identifizieren, muss man die Klasse des Empfängerobjekts kennen, also die Klasse des Werts der Variable, auf der die Methode aufgerufen wurde. Die ist aber in der Regel nur auf Basis einer vollständigen Programmanalyse bestimmbar, die sich nicht lokal durchführen lässt. Das Lokalitätsprinzip wird also durch das dynamische Binden weiter aufgeweicht als durch den Unterprogrammaufruf allein.

Dieser Umstand hat dazu geführt, dass das dynamische Binden von Skeptikerinnen und Gegnerinnen der objektorientierten Programmierung schon als eine Art Goto der 90er Jahre betrachtet wurde. Dieser Vergleich ist jedoch nicht ganz fair, weil, genau wie beim statisch gebundenen Unterprogrammaufruf, die Aufruferin ja gar nicht wissen muss, welche genauen Anweisungen als Antwort darauf

dynamisches Binden als Goto der 90er Jahre?



⁹⁰ Jede, die einmal ein größeres objektorientiertes Programm debuggt hat, weiß, wovon ich spreche: Im Single-step-Modus der IDE springt der Programmzähler wild zwischen den verschiedenen Methoden und deren Klassen hin und her und häufig ist es schon nach kurzer Zeit kaum mehr möglich, zu rekonstruieren, wie man an die Stelle gekommen ist, an der man sich gerade befindet. Viele Programmiererinnen ertappen sich dann dabei, dass sie, wie in der Steinzeit der Programmierung, Print-Anweisungen in ihre Programme einbauen, die den Programmablauf in nicht-flüchtiger Form festhalten. Zahlreiche Frameworks bieten darüber hinaus Tracing- oder Logging-Funktionen, mit denen es mit vergleichsweise wenig Aufwand möglich ist, den Programmablauf aufzuzeichnen und zu rekonstruieren. Aus der Betrachtung des Programmtexts ist dies nämlich meistens unmöglich.



ausgeführt werden müssen — es reicht, zu wissen, welchen Vertrag die aufgerufene Methode (das aufgerufene Unterprogramm) erfüllt. Dies sollte nach den Regeln des Subtyping (Abschnitt 54.2) stets unabhängig vom vertragerfüllenden Objekt sein.

Andererseits sind die Verträge in der Praxis gar nicht im Programmtext spezifiziert, oder kennen Sie ein Programm, in dem für jede dynamisch gebundene Methode Vor- und Nachbedingungen spezifiziert wären? Daher kann es bei der Betrachtung der Aufrufstelle sehr wohl interessant sein, was denn nun genau in der aufgerufenen Methode passiert ist, z. B. weil man sich eine bestimmte, resultierende Variablenbelegung nicht erklären kann. In diesen Fällen wird man sich also, beim Tracen oder beim Debuggen, auch den aufgerufenen Code anschauen wollen. Das Problem ist nur, dass man gar nicht weiß, an welcher Stelle man schauen muss. Es bleibt in der Praxis also nur, das Programm erneut auszuführen, vor dem dynamisch gebundenen Aufruf zu stoppen und sich den Variableninhalt anzusehen oder den Programmablauf Schritt für Schritt zu verfolgen, mit all den oben beschriebenen Problemen.

Es ist wohl unbestritten, dass objektorientierte Programme schwerer zu tracen und zu debuggen sind als prozedurale. Wie schon bei den statisch gebundenen Unterprogrammaufrufen (die ja ebenfalls ein Problem darstellen können) ist die Frage jedoch, ob das, was man durch das dynamische Binden hinzugewinnt, den Preis aufwiegt. Während diese Frage jede für sich selbst entscheiden muss, so scheint die Antwort für viele Programmiererinnen — weit mehr als eine Generation nach dem Aufkommen der Objektorientierung und damit vor dem Hintergrund genügend praktischer Erfahrung — überwiegend positiv zu sein. Man darf aber auch die Neinsagerinnen nicht als Ewiggestrige abstempeln — sie mögen gute Gründe haben.

Fazit

57 Das Problem der eindimensionalen Strukturierung

Klassen sind die Module der objektorientierten Programmierung. Die Menge der Module und damit das Programm werden durch die Vererbungshierarchie weiter strukturiert. Dazu parallel (und weitgehend, bis auf die Vererbung von Beziehungen unabhängig) gibt es noch eine Struktur, die durch das Bestehen von *Beziehungen* zwischen Klassen (genauer: das Bestehen der Möglichkeit von Beziehungen zwischen Objekten der Klassen; s. Kapitel 2 in Kurseinheit 1) geprägt ist — diese ist aber nicht hierarchisch und insgesamt eher unorganisiert, weswegen sie nicht zur systematischen Programmorganisation taugt. Außerdem besteht ein gewisser Konflikt zwischen der Hierarchie der Subklassenbeziehung und den Beziehungen zwischen Objekten: Wenn man einen Teilbaum der Vererbungshierarchie herauslöst, trennt man damit praktisch immer Beziehungen zwischen Mitgliedern des Teilbaums und anderen auf. Die Klassenhierarchie stellt also insbesondere keine Form der *hierarchischen Modularisierung* dar.



Insbesondere bei größeren Programmen kann leicht das Verlangen auftreten, ein Programm nach mehreren Kriterien gleichzeitig zu strukturieren. Dies kann beispielsweise verschiedene Vererbungshierarchien betreffen — so, wie man in der Biologie eine Taxonomie der Arten nach Herkunft (Genetik) und nach Merkmalen erstellen kann und, da beide ihren Nutzen haben, man weder auf die eine noch auf die andere ohne Not verzichten möchte, so kann man ein Programm beispielsweise unter dem Gesichtspunkt der Vererbung von Funktionalität und von Daten alternativ strukturieren wollen. Voraussetzung hierfür ist allerdings, dass die verschiedenen Darstellungen getrennt voneinander gepflegt werden und dass durch sie weder Inkonsistenzen im Code noch ungewollte Interferenzen entstehen können.

Ein Nachteil der objektorientierten Programmierung (wie auch aller anderen heute bekannten Programmierparadigmen) ist sicherlich, dass die

**viele Ansätze —
(noch) keine Lösung**

sog. **Trennung der Belange**, besser bekannt als die **Separation of concerns**, nur unzureichend unterstützt wird. Ansätze wie das Subject- oder Aspect-oriented programming wurden zwar hoch gehandelt, sind aber dennoch nicht im Mainstream angekommen. Die Gründe dafür mögen vielfältig sein aber letztlich ist es wohl immer illusorisch, zu versprechen, man könne die essentielle Komplexität, die einem Problem innewohnt, durch programmiersprachliche Mittel beseitigen. Insbesondere die getrennte Spezifikation eines Systems aus verschiedenen Sichten verlagert die Komplexität nur in das Zusammenführen der Sichten: Wie so oft muss das Ganze mehr sein als die Summe seiner Teile, um seinen Zweck zu erfüllen. Unsere heutigen Softwaresysteme sind die kompliziertesten technischen Artefakte, die die Menschheit jemals hervorgebracht hat, und wer hier Einfachheit verspricht, soll sich schämen.



Everything should be made as simple as possible — but no simpler.

58 Das Problem der mangelnden Kapselung

Als man mit der objektorientierten Programmierung begann, war man glücklich, weil man glaubte, mit dem Klassenbegriff eine natürliche Art der **Kapselung** (engl. encapsulation) gefunden zu haben, die zudem noch mit der hochangesehenen Theorie der abstrakten Datentypen in Einklang steht (zumindest einigermaßen): Klassen ergeben sich auf natürliche Weise aus der Anwendungsdomäne (als Repräsentanten von *Allgemeinbegriffen*; s. Kurseinheit 1, Kapitel 7) und Daten sowie Implementierungsdetails lassen sich hinter der *Klassenschnittstelle* (dem Protokoll der Objekte) verbergen.

Die erste große Enttäuschung kam, als man merkte, dass die ebenfalls gefeierte Vererbung die Kapselung von Klassen auf unangenehme Weise aufbrach: Wie in Kapitel 55 bemerkt, erzeugt die Vererbung starke Abhängigkeiten (auch zwischen den Implementierungsdetails!) von Klassen und ihren Subklassen. Diese Abhängigkeiten explizit zu machen vermag zwar vor Programmierfehlern zu

**starke
Abhängigkeiten
durch Vererbung**



schützen, sie kann aber die Abhängigkeiten nicht beseitigen — sie dokumentiert sie lediglich. Die Abhängigkeiten zu beschränken bedeutet wiederum, einen Teil der Ausdrucksstärke und Flexibilität der objektorientierten Programmierung aufzugeben, aber so ist das nun einmal: Alles hat seinen Preis.

Sehr viel dramatischer (und sehr viel weniger in aller Munde) ist jedoch ein ganz anderes Problem, das das gesamte bisherige Bemühen der Objektorientierung um Kapselung auszuhebeln in der Lage ist: das **Aliasing**-

Umgehung der Kapselung durch Aliase

Problem. Wenn nämlich ein Objekt, das durch ein anderes Objekt gekapselt wird, indem das andere es in einer seiner Instanzvariablen hält, einen (weiteren) Alias besitzt, der nicht selbst dem kapselnden Objekt gehört, dann nützt es nichts, wenn diese Instanzvariable von außen unzugreifbar ist⁹¹ — sie wird nämlich gar nicht gebraucht, um auf das gekapselte Objekt zuzugreifen. Man bedient sich einfach des Aliases.

In SMALLTALK wird das Problem in folgendem einfachen Codefragment klar:

```
1532 jetzt := Time now.  
1533 a := A new erzeugungszeit: jetzt. "neues Objekt mit Erzeugungszeit"  
1534 jetzt setTime: Time now "Erzeugungszeit von a ändert sich!"
```

Hier soll ein neues Objekt der Klasse A erzeugt werden und die Erzeugungszeit in einer entsprechenden Instanzvariable des Objekts festgehalten werden. Die Variable jetzt hält aber einen Alias auf das Objekt, das diese Zeit repräsentiert; ändert man dieses Zeitobjekt (wie in Zeile 1534), dann betrifft dies auch die Erzeugungszeit des Objekts a.⁹² Nun könnte man meinen, es genügte, man ersparte sich einfach den (offensichtlichen) Alias **jetzt** und schriebe stattdessen

```
1535 a := A new erzeugungszeit: Time now
```

Woher weiß man aber, dass die Methode **now** in der Klasse **Time** nicht einen Alias auf jedes neu erzeugte Objekt anlegt (beispielsweise weil **Time** Buch darüber führt, welche Instanzen es von ihr gibt) und diesen Alias nicht herausgibt oder selbst verwendet, um die Objekte zu manipulieren? Auch die alternative scheinbare Lösung,

```
1536 a := A new erzeugungszeit: jetzt copy "neues Objekt mit  
Erzeugungsdatum"
```

funktioniert aus gleichem Grunde nicht zuverlässig, denn auch **copy** kann sich (heimlich) Aliase anlegen.

Ein anderes Beispiel, bei dem Aliase fehlerhaft eingesetzt werden, ist das folgende. Angenommen, Sie wollten eine Ampelsimulation an einem Fußgängerüberweg programmieren.

⁹¹ z. B. durch eine Deklaration mit **private** oder per Sprachdefinition wie in SMALLTALK und EIFFEL

⁹² Man könnte hier argumentieren, dass Zeit ein Wert und kein Objekt und insofern auch nicht änderbar sein soll. Das ist hier aber gar nicht der Punkt.



Sie haben zwei Klassen, **Ampel** und **Leuchte**, und bauen Ihre Objekte durch folgenden Code zusammen:

```
1537 leuchteRot := Leuchte new farbe: "rot".  
1538 leuchteGelb := Leuchte new farbe: "gelb".  
1539 leuchteGruen := Leuchte new farbe: "grün".  
1540 ampel1 := Ampel new  
1541     autoRot: leuchteRot  
1542     autoGelb: leuchteGelb  
1543     autoGruen: leuchteGruen  
1544     fussgaengerRot : leuchteRot  
1545     fussgaengerGruen : leuchteGruen.  
1546 ampel2 := ampel1
```

Wenn Sie nun einem Fußgänger per

```
1547 ampel1 fussgaengerGruen an
```

grünes Licht geben wollen, gehen leider mit der einen gleich alle vier grünen Leuchten an. War das in Ihrem Sinn?

Man nennt Objekte, die die Implementierung eines Objektes ausmachen und die hinter der Schnittstelle des Objektes verborgen werden sollen,

Repräsentationsobjekte

Repräsentationsobjekte. Die Leuchten des obigen Beispiels sind allesamt Repräsentationsobjekte; sie kommen lediglich als „Innereien“ der Objekte vor, deren Repräsentation sie ausmachen. Insbesondere gibt es in obigem Beispiel keine Verwendung einer Leuchte losgelöst von einer Ampel. Dies muss aber nicht für alle Leuchten der Fall sein — es ist durchaus denkbar, dass Leuchtenobjekte in anderen oder sogar im selben Programm auch ein unabhängiges Leben (außerhalb von Ampeln) führen. Aber selbst das ist gar nicht notwendig — im gegebenen Beispiel wäre es auch denkbar, dass mit Leuchten noch einiges gemacht wird, bevor sie in eine Ampel eingebaut werden, so dass man das Aliasing-Problem weder an die Klasse **Leuchte** pauschal noch an deren Verwendung als Lieferant für Repräsentationsobjekte knüpfen kann.

Wenn Aliase also schlecht sind, dann könnte man sie ja auch einfach verbieten. Tatsächlich bieten ja Programmiersprachen wie C#, C++ und EIFFEL die Möglichkeit an, Klassen als Werttypen zu definieren, so dass bei Zuweisungen nicht automatisch Aliase entstehen. Aber durch eine derart einfache Lösung beschneidet man sich selbst nur wieder zahlreicher Möglichkeiten, wie das folgende Beispiel zeigt:

Folgen eines Alias-Verbots

```
1548 merkEsDir := Dictionary new.  
1549 a := A new.  
1550 merkEsDir at: "mein A Objekt a" put: a.  
1551 a == (merkEsDir at: "mein A Objekt a")  
1552    ifFalse: ["Oh je, du bist nicht mehr dasselbe!"].  
1553 a aendereDeinenZustand.  
1554 a = (merkEsDir at: "mein A Objekt a")  
1555    ifFalse: ["Oh je, du bist nicht mehr das gleiche!"]
```



Es ist gerade der Sinn eines Dictionaries (bzw. allgemeiner eines Containers wie einer Collection), dass keine Kopie, sondern das originale Objekt — also ein Zeiger darauf — gespeichert wird! In einer Sprache ohne Referenzsemantik wäre das jedoch nicht möglich. Man braucht also die Möglichkeit, fallweise zu unterscheiden, ob ein Objekt Aliase haben darf.

Das obige Beispiel stellt insofern kein großes Problem dar, als der fehlerhafte Umgang mit dem Aliasing durch unerwartetes Programmverhalten auffällt. Ein viel größeres Problem entsteht, wenn die Kapselung von Objekten (das Geheimnisprinzip) Gegenstand der Spezifikation eines Programms ist, die Existenz von Aliasen also mit der Spezifikation nicht vereinbar wäre. Dies ist bei allen sicherheitskritischen Anwendungen der Fall, bei denen Daten geschützt werden müssen oder Funktionen nur durch autorisierte Benutzerinnen ausgeführt werden dürfen. Gibt es dann Aliase von außen auf diese Objekte als geheime Daten- oder Funktionsträger, dann ist die Spezifikation nicht erfüllt. Am Programmverhalten ist dies jedoch nicht zu erkennen.

Geheimnisprinzip als Teil der Spezifikation

Nachdem das Problem nun hinreichend klar geworden sein sollte, was kann man dagegen tun? Zunächst einmal muss noch einmal klargestellt werden, dass die Deklaration von Instanzvariablen als von außen unzugreifbar (**private**) lediglich *Namen* verbirgt — man kann über die Schnittstelle des Objekts nicht herausfinden, *wie* es intern aufgebaut ist.

Umsetzung des Geheimnisprinzips mittels nicht zugreifbarer Variablen

Dieser Namenschutz (engl. name protection) ist das, was man häufig (mit dem Wissen vom Aliasing-Problem) mit **Geheimnisprinzip** (engl. **information hiding**) verbindet: Es verhindert, dass andere Klassen von der Existenz bestimmter Instanzvariablen abhängen, so dass diese problemlos geändert (z. B. umbenannt oder entfernt) werden können.

Das Geheimnisprinzip vermag jedoch nicht zu verhindern, dass die Repräsentationsobjekte, deren Namen verborgen werden, noch andere Namen besitzen. Dies kann z. B. immer dann der Fall sein (und ist vom Objekt, das sein Implementationsgeheimnis wahren will, kaum zu verhindern), wenn ein Objekt seine Repräsentationsobjekte bei seiner Erzeugung von außen geliefert bekommt (genau so, wie das in den obigen Beispielen in den Zeilen 1533, 1535 und 1540 der Fall war). Eine weitere Möglichkeit, die Kapselung zu durchbrechen, ist, selbst eine Referenz auf ein Repräsentationsobjekt herauszugeben, beispielsweise durch einen Getter, aber das wäre dann vom „verbergenden“ Objekt selbst zu verantworten (und zu verhindern gewesen).

Lücken

Sobald also die Möglichkeit des Aliasing besteht, ist eine echte Kapselung über das Geheimnisprinzip allein nicht mehr zur Gewährleisten. Man muss also das Aliasing irgendwie kontrollieren. Die Frage ist nur: Wie?

Eine Möglichkeit hatten wir bereits mehrfach angesprochen. Man kann das Bestreben nach Kapselung als Ausdruck des Bestehens einer *Teil-Ganzes-Beziehung* zwischen den Repräsentationsobjekten und dem Objekt, dessen Repräsentation sie ausmachen, verstehen. Die Teile sollen dabei dem Ganzen gehören in dem Sinne, dass sie nicht zugleich auch Teile anderer Objekte sein können, und darüber hinaus auch nicht von ande-

ein Lösungsansatz



ren Objekten referenziert werden können. Letzteres kann man auf einfache Weise verhindern, wenn man aus den Objekten Wertobjekte macht und die verwendete Programmiersprache keine Zeiger auf Wertobjekte erlaubt. Von den in Kurseinheit 4 und Kurseinheit 5 genannten Programmiersprachen ist das jedoch nur in C#, und da auch nur im Safe mode, möglich, nämlich wenn die „Klasse“ der Teil-Objekte per `struct` definiert wurde. Abgesehen von dieser Einschränkung ist eine solche Vorgehensweise nur selten ohne unerwünschte Nebenwirkungen — sie bedeutet nämlich immer auch, dass alle Objekte dieser „Klasse“ nur Wertobjekte sein und keine Referenzen haben dürfen, was aber die Anwendungsdomäne in der Regel nicht korrekt abbildet.

Vor diesem Hintergrund scheint der in Abschnitt 52.5.2 dargelegte Umgang EIFFELS mit Referenz- und Wertvariablen ziemlich schlau ausgedacht zu sein. Zwar erlaubt EIFFEL, auf Wertobjekte Referenzen zu haben (und somit zumindest theoretisch, dass ein Repräsentationsobjekt einen Alias besitzt), aber bei der Zuweisung einer Referenzvariable an eine Wertvariable wird immer eine (aliasfreie) Kopie des referenzierten Objekts erzeugt und zugewiesen, so dass kein Alias in die Repräsentation hinein entstehen kann. Umgekehrt wird bei der Zuweisung eines Wertobjekts an eine Referenzvariable immer eine Kopie des Wertobjekts erzeugt und die Referenz darauf angelegt. Es entsteht also faktisch kein Alias auf ein Wertobjekt, und als Wertobjekte angelegte Repräsentationsobjekte sind aliasfrei. Dumm ist nur, wenn man innerhalb der Kapsel Aliase auf Wertobjekte braucht.

die Lösung in EIFFEL

Weitergehende Mechanismen zur Aliaskontrolle in objektorientierten Programmiersprachen befinden sich derzeit alle noch in der Vorschlags- und Erprobungsphase und sollen hier deswegen nicht weiter behandelt werden.

59 Das Problem der mangelnden Skalierbarkeit

Zwar besteht jedes laufende objektorientierte Programm aus einer Menge von Objekten, jede Spezifikation eines solchen Programms besteht aber bei den heute gebräuchlichen klassenbasierten objektorientierten Programmiersprachen aus einer Menge von Klassen. Die strukturbildende Einheit der objektorientierten Programmierung auf Programmebene ist daher die Klasse. Größere Einheiten sind innerhalb der gängigsten objektorientierten Programmiersprachen nicht vorgesehen: JAVAs Packages und ähnliche Konstrukte sind allenfalls Namensräume und Einheiten der Auslieferung — der Status eines Sprachkonstrukts vergleichbar mit Klasse oder Methode kommt ihnen kaum zu.

Nun sind Klassen relativ feingranulare Gebilde. Zwar hindert einen nichts daran, große Klassen (mit Hunderten von Attributen und Methoden) zu schreiben, aber dies gilt nicht nur als schlechter Stil, es spiegelt auch die Anwendungsdomäne in aller Regel nicht angemessen wieder. Dort sind nämlich alle großen (komplexen) Dinge aus einfacheren zusammengesetzt, die, wenn sie selbst eine gewisse Komplexität haben, selbst wieder aus kleineren zusammengesetzt sind usw. Dasselbe gilt auch für die

Granularität von Klassen

Artefakte anderer Ingenieursdisziplinen: Baupläne sind in Komponenten und Unterkomponenten bzw. Systeme und Untersysteme strukturiert. Da wünscht man sich natürlich analoge Möglichkeiten in der objektorientierten Programmierung.

Nun ist es zwar möglich, Objekte mit Hilfe der Teil-Ganzes-Beziehung rekursiv aufzubauen (und im oben diskutierten Rahmen auch zu kapseln, also Teile vollständig hinter Ganzen zu verbergen), aber für Klassen gilt das nicht. Zwar ist es hier möglich, über sog. *innere Klassen* (in JAVA) Klassen zu strukturieren, aber allein schon die Tatsache, wie relativ wenig davon Gebrauch gemacht wird, zeigt, dass es sich dabei um keinen besonders nützlichen Mechanismus handelt.⁹³ Tatsächlich ist es nämlich — wie schon in Kapitel 58 angesprochen — so, dass Objekte einer Klasse nicht immer Teile von Objekten anderer Klassen sind (und schon gar nicht immer der gleichen Klassen), sondern vielmehr einzelne Exemplare (Instanzen) Teil sein und vielleicht sogar selbst Teile haben können. Man kann also die hierarchische Struktur objekt-orientierter Systeme genauso wenig auf Klassenebene vorschreiben, wie man den Aufbau einer Maschine anhand lediglich der Typen ihrer Teile (Schrauben etc.) beschreiben könnte (ohne festzulegen, wo jede einzelne Instanz genau hingehörte). Was man vielmehr bräuchte, sind **Komponenten** als zusätzliches, von Klassen und Objekten verschiedenes Programmiersprachenkonstrukt.

Leider ist es mit der Einführung von Komponenten in objektorientierte Programmiersprachen bislang noch nicht besonders weit. Das merkt man schon daran, dass keine weit verbreitete objektorientierte Programmiersprache das Schlüsselwort **component** verwendet, ja nicht einmal reserviert. Stattdessen lässt man die Programmiererinnen alles in Form von Klassen definieren und Komponenten immer zur Laufzeit, per Aggregation von Objekten, zusammenbauen. So schwache Konzepte wie Pakete (JAVA) oder Assemblies (C#) können dabei keineswegs einen Komponentenbegriff ersetzen, da sie lediglich Klassen gruppieren (und dabei auch noch ignorieren, dass dieselbe Klasse Instanzen für Komponenten verschiedenen Typs liefern kann). Es ist meine persönliche Vermutung, dass an dieser Front in den nächsten Jahren noch der größte Fortschritt erzielt werden kann.

mangelnde
hierarchische (De-)
Komponierbarkeit
von Klassen

Komponenten in
objektorientierten
Programmier-
sprachen

60 Das Problem der mangelnden Eignung

To a woman with a hammer, everything looks like a nail.

Wie alle Ingenieurinnen verfallen objektorientierte Programmiererinnen gern dem Hammerprinzip: Wenn man einen Hammer in der Hand hat, sieht alles wie ein Nagel aus. Nicht alle

⁹³ Das gilt allerdings nicht für SCALA, die „Scalable Language“ — hier werden Singleton-Klassen, Objekte genannt, zur hierarchischen Strukturierung eines Programms verwendet.



Aufgaben sind aber gleichermaßen zur Lösung per objektorientierter Programmierung geschaffen. Für viele logische und Suchprobleme sind beispielsweise *funktionale* oder *logische Programmiersprachen* weit besser geeignet; aber auch viele Batch- und Scripting-Probleme (in denen lediglich vorhandene Programme mit den richtigen Daten versorgt und angestößen werden müssen) haben eher imperativ-prozeduralen denn objektorientierten Charakter.

Auch wenn pauschale Aussagen riskant sind, so erscheinen doch Probleme, die einen hohen algorithmischen Anteil und vergleichsweise simple Datenstrukturen verlangen, weniger geeignet für die objektorientierte Programmierung. Wie schon in Kapitel 56 erwähnt, verlangt die „gute“ objektorientierte Programmierung, den Code (die Funktionalität) auf die Klassen aufzuteilen, die die Daten definieren, auf denen der Code arbeitet. Da größere Probleme in der Regel auf durch verschiedene Klassen definierte Daten zugreifen müssen, wird der Code durch seine Datenbindung regelrecht zerfleddert.

Funktion über Daten

Ein ähnlich gelagertes Problem hat man, wenn man Programme entwickelt, in denen es vor allem um Abläufe geht. Hier möchte man, dass die Reihenfolge der Schritte, die auszuführen sind, in einem Stück festgehalten wird (*Lokalitätsprinzip!*) und nicht auf zig Klassen aufgeteilt ist. In solchen Fällen steht das Interesse an der Struktur der Funktionen über dem an der Struktur der Daten — dass hier die objektorientierte Programmierung nicht ideal ist, liegt eigentlich auf der Hand.

So hat man es bei der Wahl einer geeigneten Programmiersprache in der Praxis fast immer mit einem Abwägungsproblem zu tun. Wenn man sich für die objektorientierte Programmierung entscheidet, bleibt die Organisation der Funktionen auf der Strecke, wenn man sich für die prozedurale Programmierung entscheidet, werden die Daten auf kaum nachzuvollziehende Weise hin- und hergeschickt oder sind global, was auch kein Idealzustand ist. Sprachen, die eine Mischung mehrerer Paradigmen erlauben, scheinen die Lösung zu sein. Für die Didaktik eignen sie sich jedoch weniger, schon weil sie Anfängerinnen mit ihrer großen Auswahl an Konstrukten und der unüberschaubaren Anzahl von Alternativen, wie man ein einzelnes Problem lösen kann, überfordern. C++ ist ein gutes Beispiel dafür.



Ein anderes Problem ist der Einsatz objektorientierter Programmierung in Verbindung mit relationalen Datenbanken. Zwar spiegelt ein gut entworfenes Datenbankschema, genau wie ein gut entworfenes Klassenmodell, eine Strukturierung der Anwendungsdomäne wider, doch tun es beide mit ganz unterschiedlichen Mitteln: Während relationale Datenbanken wertbasiert sind (alle Daten werden als Tupel primitiver Datentypen wie Zahlen und Zeichenketten dargestellt), sind objektorientierte Programme zeigerbasiert. Beziehungen werden in relationalen Datenbanken über die Verwendung gleicher Werte in *Schlüsseln* und *Fremdschlüsseln* sowie über Join-Operationen hergestellt, in objektorientierten Programm über Referenzen und deren *Dereferenzierung (Navigation)*. Vererbung bzw. Subtyping, für die objektorientierte Programmierung charakteristisch, gibt es in relationalen Datenbanken gar nicht.⁹⁴ Sollen also relationale Daten durch objektorientierte Programme verarbeitet werden, muss man sich an die Prinzipien der relationalen Welt anpassen und damit ein Gutteil dessen, was Objektorientierung ausmacht, aufgeben, weswegen man hier auch häufig von einem *Impedance mismatch* spricht (das entsprechende deutsche Wort „Fehlanpassung“ ist in diesem Zusammenhang ungebräuchlich).



WIKIPEDIA



Etwas anders gelagert ist der Fall, dass man eine relationale Datenbank dazu einsetzt, eine objektorientierte zu simulieren. In diesem Fall werden die Daten zunächst (wie in der gewöhnlichen objektorientierten Programmierung) angelegt und nur zu Persistenz- und Synchronisationszwecken (bei Mehrbenutzerinnensystemen) in der Datenbank abgelegt. Die Abbildung der objektorientierten Klassenstruktur auf das relationale Schema wird dabei heute meistens durch ein sog. *Persistenzlayer* erreicht — das Programm selbst muss sich um die Datenhaltung nur auf sehr abstrakter Ebene kümmern. Dennoch muss man auch hier die Frage stellen, warum man einer relationalen Datenbank den Vorzug vor einer objektorientierten gegeben hat — am Ende, weil im betrieblichen Umfeld häufig bereits relationale Datenbanken mit gutem Ergebnis verwendet werden und die Umstellung auf Objektorientierung in der Datenhaltung mit unwägbaren Risiken verbunden scheint — schließlich sind die Daten häufig der eigentliche Wert eines Softwaresystems.

Zuletzt, und beinahe paradoxe Weise, ist auch die GUI-Programmierung nicht unbedingt ein Heimspiel für die objektorientierte Programmierung.

Zwar kann man für die verschiedenen Arten von GUI-Elementen noch ganz gut Klassen angeben, die die Gemeinsamkeiten im Aussehen der in einem konkreten GUI verwendeten Objekte herausfaktorisieren, aber spätestens beim gemeinsamen Verhalten ist Schluss: Zwei Buttons beispielsweise unterscheiden sich nicht nur bezüglich ihrer Position und des angezeigten Texts, sondern auch darin, welche Aktion ausgeführt wird, wenn sie gedrückt werden. Da sich die Instanzen einer Klasse aber alle Methoden teilen, ist es nicht möglich, für verschiedene Buttons derselben Klasse verschiedene Implementierungen einer Methode anzugeben. Hier kann man lediglich versuchen, eine Indirektion einzubauen, in SMALLTALK über einen Block, der die auszuführende Methode beinhaltet, in JAVA über anonyme innere Klassen, die für eine bestimmte Methodensignatur eine Implementierung liefern, die nur den

**Einzigartigkeit von
GUI-Elementen**

⁹⁴ Dies ist strenggenommen nicht richtig: Schon der Standard SQL:1999 enthielt starke objektorientierte Anleihen. Ich bin mir jedoch nicht sicher, ob diese jemals in der Praxis angekommen sind.



Instanzen dieser (unbenannten) Klasse gehört, und in C++ sowie C# über Funktionszeiger (Delegates in C#). Von Haus aus besser geeignet scheint hier aber die prototypenbasierte Variante der objektorientierten Programmierung, wie in der Einleitung zu Kurseinheit 2 bemerkt (und wie sie ja auch in Form von JAVASCRIPT seit Jahren einen heimlichen Siegeszug feiert).

61 Lösungen zu den Selbsttestaufgaben

Selbsttestaufgabe 55.1 (Seite 306)

Lesen Sie weiter!



Kurseinheit 7: Objektorientierter Stil

There does not now, nor will there ever, exist a programming language in which it is the least bit hard to write bad programs.

Lawrence Flon

Egal, wie formal sie auch sind: Programmiersprachen sind Sprachen und erlauben einer Autorin damit, sich auf eine persönliche Art und Weise auszudrücken. Dabei bestimmt die Ausdrucksweise nicht den Inhalt des Programms (seine Funktion), sondern seine Qualität, also z. B. wie effizient ein gegebenes Problem damit gelöst wird oder wie verständlich die Formulierung der Lösung für die Betrachterin ist. Besonders die Verständlichkeit hat etwas mit Schreibstil zu tun; neben ihr spielen aber auch noch andere Parameter in Stilfragen eine Rolle, so z. B. Mode und Ästhetik (Eleganz).

So hat es fraglos in den letzten Jahrzehnten eine Wandlung in Stilfragen gegeben, und zwar weg vom mathematisch-pregnanten hin zum prosaisch-verbosen Stil. Das folgende Beispiel soll davon einen Eindruck geben:

mathematisch-
pregnanter vs.
prosaisch-verboser
Stil

```
1556 PROGRAM marriage(input,output);
1557 {Problem der stabilen Heirat}
1558 CONST n = 8;
1559 TYPE man = 1 .. n; woman = 1 .. n; rank = 1 .. n;
1560
1561 VAR m: man; w: woman; r: rank;
1562   wmr: array [man, rank] OF woman;
1563   mwr: ARRAY [woman, rank] OF man;
1564   rmw: ARRAY [man, woman] OF rank;
1565
1566   rwm: ARRAY [woman, man] OF rank;
1567   x: ARRAY [man] OF woman;
1568   y: ARRAY [woman] OF man;
1569   single: ARRAY [woman] OF BOOLEAN;
1570
1571 PROCEDURE print;
1572   VAR m: man; rm, rw: INTEGER;
1573 BEGIN rm := 0; rw := 0;
1574   FOR m := 1 TO n DO
1575     BEGIN write(x[m]:4);
1576       rm := rm + rmw[m,x[m]]; rw := rw + rwm[x[m],m]
1577     END ;
1578   writeln(rm:8, rw:4)
1579 END {print} ;
```



```

1581 PROCEDURE try(m: man);
1582   VAR r: rank; w: woman;
1583
1584   FUNCTION stable: BOOLEAN;
1585     VAR pm: man; pw: women;
1586     i, lim: rank; s: BOOLEAN;
1587   BEGIN s := TRUE; i := 1;
1588     WHILE (i < r) AND s DO
1589       BEGIN pw := wmr[m,i] ; i := i+1;
1590         IF NOT single[pw] THEN
1591           s := rwm[pw,m] > rwm[pw,y[pw]]
1592       END ;
1593       i := 1; lim := rwm[w,m];
1594     WHILE (i < lim) AND s DO
1595       BEGIN pm := mwr[w,i] ; i := i+1;
1596         IF pm < m THEN s := rmw[pm,w] > rmw[pm,x[pm]]
1597       END ;
1598     stable := s
1599   END {stable} ;
1600
1601 BEGIN {try}
1602   FOR r := 1 to n DO
1603     begin w := wmr[m,r];
1604       IF single[w] THEN
1605         IF stable THEN
1606           BEGIN x[m] := w; y[w] := m; single[w] := FALSE;
1607             IF m < n THEN try(succ(m)) ELSE print;
1608             single[w] := TRUE
1609           END
1610     END
1611
1612 END {try};
1613 BEGIN
1614   FOR m := 1 TO n DO
1615     FOR r := 1 TO n DO
1616       BEGIN read(wmr[m,r]) ; rmw[m,wmr[m, r]] := r
1617     END;
1618   FOR w := 1 TO n DO
1619     FOR r := 1 TO n DO
1620       BEGIN read(mwr[w,r]); rwm[w,mwr[w, r]] := r
1621     END;
1622   FOR w := 1 TO n DO single[w] := TRUE;
1623   try(1)
1624 END .

```

Es handelt sich dabei um ein PASCAL-Programm zur Lösung des Problems der stabilen Heirat, wie es in dem Klassiker „Algorithmen und Datenstrukturen“ von NIKLAUS WIRTH nachzulesen ist. Eine mir leider entfallene Quelle soll übrigens gesagt haben, eine Untersuchung habe hervorgebracht, dass ein bedeutender Anteil aller Variablen in Programmen „i“ heiße. Nun wurden in obigem Beispiel die Variablen nicht „i“, sondern „r“, „m“, „w“ usw. genannt, aber das grundlegende Problem bleibt das gleiche: Man muss sich schon ziemlich in das Programm bzw. den dazugehörenden Text vertiefen, um zu erfassen, wofür die Variablen stehen.



Heute ist es üblich, Bezeichner (Namen für Module, Typen, Variablen, Prozeduren und Funktionen) in einem Programm so zu wählen, dass Kommentare bzgl. der Bedeutung des jeweiligen Programmelements unnötig sind, da sie nicht viel mehr ausdrücken können, als es der Bezeichner in seinem jeweiligen Kontext ohnehin tut. Dies geht sogar so weit, dass Leute wie Kent Beck (Mitinitiator und Verfechter des sog. Extreme Programming, SMALLTALK-Veteran) meinen, ein gut geschriebenes objektorientiertes Programm bräuchte gar keine Kommentare. Dem möchte ich entgegenhalten, dass manchmal die Lösung eines Problems in seiner verständlichsten Form um vieles uneleganter ist als eine, die mit einer gewissen Raffinesse daherkommt, sich dafür aber nicht jeder unmittelbar erschließt. In solchen Fällen ist die Versuchung groß, sich für die geistreichere Variante zu entscheiden und sie, für diejenigen Leserinnen, die einer nicht auf Anhieb folgen können, mit einem erklärenden Kommentar zu versehen. Nicht zuletzt sind es ja gerade die alles andere als offensichtlichen Algorithmen, die ihren Autorinnen zu Berühmtheit verholfen haben, und wer würde nicht gern hier und da eine eigene Marke setzen. Kryptische Namen zu verwenden ist jedoch niemals ein Zeichen von Genialität.

Wir kommen also gleich zur Stilregel Nummer 1 der objektorientierten Programmierung.

62 Namenwahl

Immer sprechende Namen verwenden.

Gegen die Verwendung langer, sprechender Bezeichner kann man einwenden, dass der Programmtext dadurch übermäßig lang wird. Anweisungen, die sonst in eine Zeile gepasst hätten, müssen u. U. mehrfach umgebrochen werden, was die Lesbarkeit nicht gerade erhöht. Auch hört man hier und da, dass lange Namen für die Programmiererin zusätzliche Schreibarbeit bedeuten. Letzteres Argument kann man jedoch kaum gelten lassen, da die meisten Entwicklungsumgebungen über eine automatische Vervollständigungsfunktion verfügen, die einem das Tippen abnimmt (und damit auch Tippfehler aufdeckt oder vermeidet). Das erste Argument ist schon schwieriger zu entkräften: Natürlich sind prägnante Namen geschwätzigen vorzuziehen und auch in der Programmierung liegt die Würze in der Kürze — insbesondere sind lange Namen, die sich nur geringfügig unterscheiden (und das auch noch wenig offensichtlich), zu vermeiden. Als Faustregel ist ein Name dann gut gewählt, wenn man alle Ausdrücke, in denen er vorkommt, schnell verstehen kann (und nicht nur, aufgrund falscher Assoziationen und Vermutungen, zu verstehen glaubt). Eine sorgfältige Programmiererin wird sich also häufiger dabei beobachten, wie sie über einen passenden Namen für ein Programmelement länger nachsinnt. Diese Zeit ist jedoch gut investiert.



62.1 Verwendung von Abkürzungen

Abkürzungen sind nicht grundsätzlich zu vermeiden — im Gegenteil, wenn sie etabliert sind und man davon ausgehen darf, dass eine Leserin des Programms sie kennt, ist ihre Verwendung (aus oben angeführten Gründen gegen zu lange Namen) sogar angezeigt. Auf hausgemachte Abkürzungen, deren Bedeutung man nur selbst kennt, sollte man hingegen verzichten.

Bei der Programmierung mit JAVA und anderen typisierten objektorientierten Programmiersprachen begegnet man häufig dem Phänomen, dass Typen und Variablen gleich heißen, sich nur in der Groß- bzw. Kleinschreibung ihres Anfangsbuchstabens unterscheiden. Ein typisches Beispiel dafür ist das folgende:

**Abkürzungen bei
Namensgleichheit
von Variable und Typ**

```
1625 Iterator iterator = list.iterator();
```

In diesen Fällen, wenn es keinen besseren Namen für die Variable gibt, ist es vollkommen legitim, eine Abkürzung für den Variablenamen zu wählen, insbesondere dann, wenn die Sichtbarkeit der Variable auf die unmittelbare Umgebung der Deklaration beschränkt ist:

```
1626 Iterator iter = list.iterator();
```

oder

```
1627 Iterator i = list.iterator();
```

sind also legitim. Sobald dies jedoch nicht der Fall ist (typischerweise schon bei der Deklaration von Instanzvariablen), sollte der lange Name bevorzugt werden. Dies gilt auch für den Fall, dass der Typ der Variable aus dem Kontext abgeleitet werden kann und deswegen nicht mehr angegeben wird (*Typinferenz*).

62.2 Namenskonventionen

Häufig findet man in einzelnen Projekten und nicht selten in ganzen Firmen Namenskonventionen vor, an die sich alle halten sollten. Namenskonventionen erleichtern nicht nur die Bezeichnerwahl (indem die Programmiererin sich an bestimmte Regeln halten kann, wird ihre schöpferische Freiheit eingeschränkt, was man durchaus auch als Entlastung empfinden kann), sie erleichtern auch das Lesen, weil die Leserin, die die Konventionen kennt, die Bedeutung des Bezeichners bzw. des dahinterstehenden Programmelements leichter entschlüsseln kann und sie sich somit schneller zurechtfindet. Allerdings ist es dazu notwendig, dass die Namenskonventionen genau festgeschrieben sind und dass sich alle darauf einigen — wenn nämlich jede ihre individuelle Auslegung der Regel hat, dann kann eine (vermeintliche) Namenskonventionen mehr Verwirrung stiften als nutzen.



It would be a mistake to protest against the rules ... on the grounds that they limit developer creativity. A consistent style favors rather than hampers creativity by channeling it to where it matters. A large part of the effort of producing software is spent reading existing software and making others read what is being written. Individual vagaries benefit no one; common conventions help everyone.



Bertrand Meyer

62.2.1 Mechanische (syntaktische) Namenskonventionen

Eine gängige Namenskonvention ist beispielsweise, Namen von Interfacetypen mit einem großen „I“ beginnen zu lassen. Andere Namenskonventionen verlangen, dass Bezeichner, die für ein Objekt oder einen Wert stehen (also Variablen und Funktionsnamen) den Typ dieses Objekts oder Werts widerspiegeln — die sog. *ungarische Notation*, von der es allerdings verschiedene Auslegungen gibt. Nach einer, eher dümmlichen, Auslegung müssen beispielsweise alle Variablen, die Strings bezeichnen, mit „str“ beginnen (eine Information, die Compiler und IDE aber ohnehin bereithalten und die deswegen nicht noch den Namen belasten muss). Nach einer sinnvolleren Auslegung sollten Variablennamen um die Verwendung ihres so bezeichneten Inhalts ergänzt werden, also die Funktion des durch sie bezeichneten Objekts oder Werts innerhalb des Kontextes, in dem die Variable gültig ist, angeben. Diese kontextbezogene Funktion kann in der objektorientierten Programmierung jedoch auch durch die Verwendung eines Interfaces anstelle einer Klasse als Typ bei der Deklaration der Variable ausgedrückt (und somit auch vom Compiler überprüft) werden.



WIKIPEDIA

62.2.2 Grammatikalisch-inhaltliche (semantische) Namenskonventionen

Überaus angemessen, wenn auch nicht immer in letzter Konsequenz einzuhalten, ist, die verschiedenen Wortarten einer natürlichen Sprache für verschiedene Arten von Programmelementen zu verwenden. So legt beispielsweise der in Kurseinheit 1, Kapitel 7 beschriebene Zusammenhang zwischen Klassen und Allgemeinbegriffen nahe, dass man für Klassennamen Substantive verwendet. Tatsächlich ist es eine vielzitierte objektorientierte Technik, in der Analysephase eines Projekts alle Substantive der Spezifikation zu extrahieren, um auf der Basis der so gewonnenen Liste die Menge der Klassen eines Systems zu identifizieren.

Klassen

Methoden, die eine Aktion implementieren (*Befehle* in EIFFEL, s. Kurseinheit 5, Abschnitt 52.2), wird man aufgrund ihres prädiktiven Charakters mit Verben benennen, wobei es eine Stilfrage ist, ob man die Infinitiv- oder die Imperativform (im Englischen übrigens kein Unterschied in der Erscheinungsform) bevorzugt. Persönlich fühle ich mich hier an keine Regel gebunden außer an die, dass Ausdrücke durch meine Namenswahl möglichst lesbar werden. So klingt

Methoden



1628 **problem loesen**

(Infinitivform) in meinen Ohren besser als

1629 **problem loese**

(Imperativform),

1630 **ausgabegeraet drucke: einenString**

(Imperativform) klingt dagegen besser als

1631 **ausgabegeraet drucken: einenString**

(Infinitivform). Man könnte natürlich der imperativen Form ein Reflexivpronomen hinzufügen wie etwa in

1632 **problem loeseDich**

aber das ist eher unüblich (obwohl nicht ohne Charme!). Verbergänzungen wie Präpositionen verwendet man in SMALLTALK dauernd (schon um mehrere Parameter voneinander abzusetzen); in Sprachen wie JAVA fügt man einem allgemeinen (und häufig überladenen) Verb dann gelegentlich noch ein Substantiv als Objekt des Prädikats hinzu, wie in

1633 **vector.addElement(anElement);**

Gerade dieses Beispiel ist jedoch nicht unumstritten, da „Element“ hier gewissermaßen redundant ist — wenn es mehrere Methoden namens „add“ gibt, kann man sie auch mittels ihrer Parametertypen unterscheiden (also *überladen*). So heißt die entsprechende Methode im JDK heute auch nur noch **add(.)**.

Keine Verben, sondern Adjektive (oder Kopula plus Prädikatsnomen) verwendet man hingegen für Methoden, die eine Abfrage darstellen (Queries; s. Abschnitt 52.2), wie etwa

1634 **stapel voll**

bzw.

1635 **stapel istVoll**

oder

1636 **menge hatElement: einElement**

Für Instanzvariablen verwendet man unterschiedliche Wortarten, und zwar abhängig davon, ob eine Instanzvariable ein Attribut oder eine Beziehung repräsentiert. Wenn es sich um ein Attribut handelt, das eine mehrwertige Qualität ausdrückt (wie Größe, Farbe etc.), dann wird man den Namen der Qualität verwenden und damit ein Substantiv (ggf. in Kleinschreibung). Wenn es sich um ein zweiwertiges (Boolesches) Attribut

| **Instanzvariablen** |



handelt, dann nimmt man das entsprechende Adjektiv (wie etwa **leer**), ein Gerundivum (z. B. **laufend**) oder ein Partizip (wie etwa **geloest**). Für Instanzvariablen, die Beziehungen ausdrücken, nimmt man gerne den Namen der Gegenrolle, also beispielsweise **mutter** in einer Kind-Mutter-Beziehung. Bei *:n*-Beziehungen nehme ich persönlich gern den Plural, also z. B. **kinder** (statt **kind**) für die umgekehrte Richtung.

Interfaces sind zwar wie Klassen Typen, aber bezeichnen keine Allgemeinbegriffe, sondern eher *Rollen*, die die Objekte, die konkrete Ausprägungen der Allgemeinbegriffe sind, spielen können. Rollen werden aber, genau wie Allgemeinbegriffe, häufig durch Substantive bezeichnet: „Mutter“ ist ein Beispiel hierfür. Andere Rollen, insbesondere die, die mit Parametern von Methoden verbunden sind, werden häufig durch Adjektive bezeichnet: **Druckbar** beispielsweise könnte der Parametertyp einer Methode **drucken** sein, den das zu druckende Objekt haben muss. Tatsächlich enden viele der gebräuchlichen Interfacenamen im Englischen auf „able“ oder „ible“, so z. B. bei **Serializable**.

Interfaces

Eine ganz interessante Option ergibt sich übrigens für Programmiererinnen, deren Muttersprache nicht Englisch ist: Man hat hier die Möglichkeit, bei der Wahl der Bezeichner zwischen zwei Sprachen zu wählen und damit eine zusätzliche Form der Differenzierung einzusetzen. Ich persönlich verwende dann gerne für Begrifflichkeiten aus der Anwendungsdomäne (also dem Gegenstandsbereich, mit dem sich das Programm befasst) deutsche Bezeichner und für solche aus der technischen Umsetzung (Hilfsklassen etc.) englische. Alternativ kann man natürlich auch alle selbst beigesteuerten Programmelemente auf Deutsch benennen, um sie von den aus Bibliotheken und Frameworks zusammengeklaubten zu unterscheiden.

gezielter Einsatz der Muttersprache

63 Formatierungskonventionen

Neben den reinen Namenskonventionen einigt man sich häufig auch auf Formatierungskonventionen (zusammen mit Namenskonventionen und anderen Richtlinien englisch als *coding conventions*, deutsch oft auch, und etwas zu allgemein, als *Programmierstil* bezeichnet). Diese regeln so Dinge wie Einrückungen und an welchen Stellen Zeilenumbrüche, Leerzeilen und Leerzeichen einzufügen sind. Sie dienen damit ebenfalls der besseren Lesbarkeit.

White space contributes as much to the effect produced by a software text as silence to the effect of a musical piece.

Bertrand Meyer

Formatierungskonventionen sind aber nicht nur eine Richtschnur für die Einzelne, die sich vielleicht unsicher ist, ob bzw. wo sie ein Leerzeichen einfügen soll — sie vermeiden auch den Effekt, dass jede Programmiererin im Team ihre eigenen Vorlieben pflegt, was in der Spalte dazu führt,

Formatierungskonventionen zur Vermeidung unproduktiver Anpassungen

dass wenn eine Programmiererin den Code einer anderen anfasst, sie zunächst einmal damit beginnt, diesen so zu formatieren, dass er ihren Lesegewohnheiten entspricht. Die ursprüngliche Autorin reagiert natürlich empört und wird nichts anderes tun, als dies bei nächster Gelegenheit wieder rückgängig zu machen, sprich zu ihren eigenen Vorlieben zurückzukehren. Dies ist alles andere als produktiv.

Ein wirksames Gegenmittel gegen solch Energieverschwendungen sind automatische Codeformatierer, die auf Knopfdruck bestimmte Codierungs-konventionen umsetzen. Wo immer verfügbar, sollte man sich zur Angewohnheit machen, diese auch einzusetzen, selbst wenn man allein arbeitet, schon um sich mit der Entwicklung des persönlichen Programmierstils nicht zu weit von dem, was allgemein üblich ist, zu entfernen. Die Programmierung ist nicht geeignet, Individualität zum Ausdruck zu bringen, geschweige denn, sie voll auszuleben.

**automatische
Codeformatierer**

64 Kurze Methoden

Wer sich den Quellcode objektorientierter Programme ansieht, der wird auffallen, dass die Methoden im Mittel ziemlich kurz sind. Wie bereits in Kurseinheit 6, Kapitel 56 erwähnt, ist dies Folge des Umstandes, dass in der objektorientierten Programmierung die Funktionalität auf Basis der Daten, von denen sie abhängt, aufgebrochen und aufgeteilt wird. Sobald eine Funktion verschiedenartige Daten manipuliert (also Objekte, die Instanzen verschiedener Klassen sind), ist es wahrscheinlich, dass diese Funktion nicht zur Gänze in einer Methode implementiert wird.

Was hier zunächst wie eine unmittelbare Folge des objektorientierten Paradigmas erscheint, hat sich zu einem objektorientierten Stil weiterentwickelt: Eine typische objektorientierte Programmiererin scheut sich nicht, Methoden zu schreiben, die nur aus einer Zeile bestehen (oder die nur eine Anweisung, wenn auch mit geschachtelten Ausdrücken, enthalten) — im Gegenteil, sie fühlt sich sogar gut dabei, denn was sie da gerade produziert, gilt als objektorientierter Stil. So ist es sogar üblich, Teile einer Methode in eine neue auszulagern (das Extract-method-Refactoring, das einige vielleicht aus ECLIPSE und ähnlichen Entwicklungsumgebungen kennen), auch wenn dieser Teil (zunächst) ausschließlich von seiner ursprünglichen Position aus aufgerufen wird, wenn es nur der besseren Lesbarkeit dient (also insbesondere Wiederverwendung keine Rolle spielt). Ein positiver Begleiteffekt dieser starken Zergliederung von Funktionalität ist die hohe Dichte an Bezeichnern in objektorientierten Programmen: Da jede Teilfunktion, die in eine Methode ausgelagert wird, einen eigenen, eindeutigen (bis auf Überladen/Überschreiben) Namen haben muss, wird die Programmiererin dazu gezwungen, sich ständig (in Form der Namenswahl für Bezeichner) dazu zu äußern, was sie gerade macht.

**mehr Methoden
bringen bessere
Dokumentation**

Link



65 Deklarativer Stil

Einhergehend mit kurzen Methoden und sprechenden Bezeichnern ist ein deklarativer Programmierstil für die objektorientierte Programmierung typisch: Die Ausdrucksform bemüht sich mehr um das Was als um das Wie. Der Effizienzgedanke ist dabei sekundär — mögliche Optimierungen werden dem Compiler überlassen und ansonsten für später aufgehoben, wenn sich herausstellen sollte, dass die Abarbeitung einer deklarativ formulierten Lösung zu ineffizient ist.

Da die objektorientierte Programmierung aber ihrem Wesen nach eher imperativ als deklarativ ist, wird sich das Deklarative im wesentlichen auf den Aufruf von Methoden beschränken, die nach dem benannt sind, was sie tun. So ist es in der objektorientierten Programmierung durchaus üblich, einzelne Schleifen, in denen beispielsweise ein Element gesucht wird, aus einem Methodenrumpf in eine eigene Methode zu verbannen und durch einen entsprechenden Methodenaufruf zu ersetzen. Das Programm liest sich also nur deklarativ und ist es nicht wirklich — es handelt sich ja auch nur um einen Stil.

Ein Beispiel für einen deklarativen Programmierstil geben die folgenden Gegenüberstellungen (Atome und Literale sind hier Konzepte aus der Aussagenlogik):

```
1637 auswerten
1638   ^ atom auswerten = negiert not
```

(deklarativ) in einer Klasse **Literal** mit Instanzvariable **atom** anstelle von

```
1639 auswerten
1640 negiert
1641   ifFalse: [^ atom auswerten]
1642   ifTrue: [^ atom auswerten not]
```

(imperativ) oder

```
1643 auswerten
1644   ^ (literale collect: [ :l | l auswerten]) includes: true
```

(deklarativ) in einer Klasse **Klausel** mit Instanzvariable **literale** anstelle von

```
1645 auswerten
1646   | answer |
1647   answer := false.
1648   literale do: [ :l | answer := answer or: l auswerten]
1649   ^ answer
```

(imperativ). Es dauert eine Weile, bis man sich das Imperative abgewöhnt hat, aber es lohnt sich.

Für beide Alternativen der Methode **auswerten** in Klasse **Klausel** gibt es übrigens eine Shortcut-Variante (die so heißt, weil die Iteration ggf. vorzeitig abgebrochen wird):



```
1650 auswerten
1651   literale detect: [ :l | l auswerten] ifNone: [^ false].
1652     ^ true
```

bzw.

```
1653 auswerten
1654   literale do: [ :l | l auswerten ifTrue: [^ true]]
1655     ^ false
```

aber solange man sich nicht sicher ist, dass eine (vermeintliche) Abkürzung funktional äquivalent ist (also dasselbe Ergebnis liefert), sollte man von solchen Optimierungen die Finger lassen. (Es könnte beispielsweise sein, dass **auswerten** für Literale einen *Seiteneffekt* hat; in diesem Fall wären die optimierten Versionen nicht mehr äquivalent!)

We should forget about small inefficiencies, say about 97% of the time:
premature optimization is the root of all evil.

Tony Hoare/Don Knuth

Ein anderes Zeichen eines deklarativen Programmierstils ist die Verwendung von Zusicherungen (Vor- und Nachbedingungen, Invarianten) anstelle von Kommentaren. Anstatt also umständlich zu formulieren

**Verwendung von
Zusicherungen
anstelle von
Kommentaren**

```
1656 pop
1657   "der Empfänger darf nicht leer sein"
1658   ...
```

schreibt man besser

```
1659 pop
1660   assert: [self empty not].
1661   ...
```

wenn es die Sprache zulässt. Damit schlägt man zwei Fliegen mit einer Klappe: Man kann die Zusicherungen zur Laufzeitverifikation einsetzen und man zeigt dem Aufrufer der Methode, wie er selbst prüfen kann, ob die Vorbedingung eingehalten ist.

66 Der Bibliotheksgedanke

Ein Großteil des Erfolgs der objektorientierten Programmierung hängt an der Verfügbarkeit und der systematischen Verwendung von Bibliotheken. In SMALLTALK ist das selbstverständlich, da hier Sprache und Bibliothek praktisch nicht zu trennen sind. Aber auch in JAVA ist ein Teil der Sprachdefinition in bestimmte, spezielle Klassen wie **Object**, **Thread** und **Throwable** verlagert (vgl. Kapitel 47). Auch kommt praktisch kein JAVA-Programm ohne



die Verwendung bestimmter Bibliotheksklassen aus — man denke nur an die allgegenwärtige Verwendung des Collection-Frameworks, ohne das Programmiererinnen zur ständigen Abfassung ewig gleichen Codes verdammt wären.

Zum objektorientierten Programmierstil gehört es, eine Lösung eines konkreten Problems möglichst umfassend aus existierenden, idealerweise verbreiteten und bewährten Bibliotheken zusammenzuklauben. Jede nicht selbst geschriebene Programmzeile ist ein Gewinn, jede Implementierung einer noch so kleinen Funktion, die es bereits in irgendeiner Bibliothek gibt, ist ein Verlust. Dabei sind die Vorteile der Verwendung von Bibliotheken mannigfaltig: In der Regel können Sie voraussetzen, dass die Implementierungen korrekt sind (und Probleme und Sonderfälle berücksichtigen, an die Sie im Traum nicht gedacht hätten), die Wartung und Anpassung übernehmen andere für Sie und nicht zuletzt dürfen Sie bei weit verbreiteten Bibliotheken voraussetzen, dass deren Funktionalität auch anderen Programmiererinnen bekannt ist, so dass die Verwendung einer Bibliotheksklasse keiner weiteren Erklärung bedarf. Ihr eigener Beitrag wird dadurch klein und überschaubar, was man Ihnen in keinem Fall als Faulheit oder Arbeitsverweigerung auslegen sollte⁹⁵, sondern als wahre Größe: Sie kennen das Werk anderer, Sie wissen es zu schätzen und zu nutzen.

so wenig wie
möglich selbst
machen

67 Ausgewogene Verteilung

Ein weiteres Kennzeichen der objektorientierten Programmierung ist, dass Klassen nicht ins Uferlose wachsen. Wenn der Methodenumfang einer Klasse immer weiter ansteigt, wird die erfahrene objektorientierte Programmiererin bald den Verdacht schöpfen, dass es sich bei der Klasse in Wirklichkeit nicht um eine, sondern um mehrere handelt. Dafür gibt es zwei Erklärungen:

1. Die Klasse steht nicht für *eine* Abstraktion der Anwendungsdomäne, sondern für mehrere. In diesem Fall sollte die Aufteilung der Klasse in mehrere — eine für jede Abstraktion — leicht fallen: Man ordnet zunächst die Daten den Abstraktionen (Allgemeinbegriffen) zu und lässt dann die Methoden den Daten folgen.
2. Die Klasse steht zwar für *eine* Abstraktion der Anwendungsdomäne, aber dies auf einem höheren Abstraktionsniveau als das der Implementierung, die Sie gerade betrachten. Dafür gibt es wiederum mindestens zwei mögliche Erklärungen:

⁹⁵ Ganz im Gegenteil: Ich würde es als Unterlassung ansehen, wenn sich jemand weigert, sich erst in Bibliotheken umzusehen, bevor sie mit der Entwicklung eigener Ideen beginnt.



- a. Die Abstraktion ist eine *Generalisierung* (s. Kurseinheit 1, Abschnitt 9.1) und Sie haben all deren *Spezialisierungen* in einer Klasse zusammengefasst. In diesem Fall müssen Sie lediglich die unterschiedlichen Spezialisierungen identifizieren und die jeweils darauf bezogenen (dafür charakteristischen) Daten und Funktionen in neu zu schaffenden Subklassen verlagern. Lediglich das allen Fällen gemeinsame Protokoll verbleibt dann in der (idealerweise abstrakten) neuen Superklasse. Ein guter Indikator für diesen Fall ist das wiederholte Vorkommen *gleicher Fallunterscheidungen*, insbesondere dann, wenn diese Fallunterscheidungen die Art der Objekte betreffen (vgl. dazu auch das sog. Replace-conditional-with-polymorphism-Refactoring).
- b. Die Abstraktion ist eine *Aggregation* (oder *Komposition*, s. Kurseinheit 1, Abschnitt 2.3), also eine Zusammensetzung eines Ganzen aus mehreren Teilen. In diesem Fall müssen Sie die Teile als logische Einheiten identifizieren und dafür neue Klassen formulieren. Für den Fall, dass diese neuen Klassen außerhalb der Abstraktion keine Bedeutung haben, können Sie in Erwägung ziehen, sie als innere Klassen zu deklarieren (wenn Ihre Programmiersprache das erlaubt), um so den Namensraum nicht zu überfrachten und die von anderen wahrgenommene Zahl der Klassen nicht unnötig zu vergrößern.



68 Das Gesetz Demeters (Law of Demeter)

Eine der klassischen Arbeiten zum Thema objektorientierter Programmierstil ist die zum sog. **Law of Demeter** von Lieberherr, Holland und Riel. DEMETER ist ein CASE-Werkzeug, das selbst kaum weiter Verbreitung gefunden hat und das wohl heute weitgehend unbekannt wäre, wenn nicht eben dieses Gesetz Demeters es zu einiger Bekanntheit gebracht hätte.



68.1 Bedeutung

Das Gesetz Demeters besagt nicht mehr, als dass Nachrichten nur an Objekte versendet werden dürfen, die der Sender selbst kennt oder erzeugt. Dabei ist Kennen ganz im Sinne von Kurseinheit 1 zu verstehen: Ein Objekt kennt ein anderes, wenn es in direkter Beziehung dazu steht, wenn es also auf eine Variable direkt (also ohne den Umweg über ein anderes Objekt) Zugriff hat, die das andere Objekt benennt. Es kennt das andere Objekt dauerhaft (oder zumindest für eine längere Dauer), wenn es sich bei der Variable um eine Instanzvariable handelt, und temporär, wenn es sich bei der Variable um den formalen Parameter einer Methode handelt (wobei die Dauer des Kennens dann auf die Dauer der Abarbeitung der Methode beschränkt ist, es sei denn, der formale Parameter wird einer Instanzvariable zugewiesen). Temporäre (lokale) Variablen werden beim Gesetz Demeters nicht mitgezählt.

Das Gesetz Demeters wird typischerweise verletzt, wenn Nachrichten an Objekte gesendet (Methoden auf Objekten aufgerufen) werden, die selbst nur als Ergebnis eines Nachrichtenausdrucks (Methodenaufrufs)

typische Ursache der Verletzung des Gesetzes



vorliegen. Dies ist in der Regel bei Kettenaufrufen der Fall (s. u.), kann aber auch über eine zwischenzeitliche Zuweisung zu einer temporären Variable erfolgen.

Sinn und Zweck des Gesetzes Demeters ist, die Kopplung und damit die Entwurfsabhängigkeiten zwischen Klassen zu verringern. Wird das Gesetz Demeters verletzt, kann die Änderung (des Protokolls) einer Klasse dazu führen, dass auch Klassen angepasst werden müssen, die selbst in keiner unmittelbaren Beziehung zu der Klasse stehen (sondern eben nur in einer mittelbaren, die nach dem Gesetz vermieden werden soll).

Ziel des Gesetzes

Das Gesetz Demeters wird oft (und leicht verkürzt) in folgender Phrase zusammengefasst:

Das Gesetz Demeters: Sprich nicht mit Fremden.

Bezogen auf SMALLTALK heißt das, dass Methodenaufrufe zwar geschachtelt, aber nicht verkettet erfolgen dürfen:

1662 `a doX`

ist also erlaubt, wenn `a` eine Instanzvariable oder ein formaler Parameter ist, ebenso

1663 `a doX: b doY`

wenn für `b` dasselbe gilt, nicht jedoch

1664 `a doX doY`

oder

1665 `| b |`
1666 `b := a doX.`
1667 `b doY`

Man beachte, dass Demeters Gesetz faktisch eine neue, kontextabhängige *Zugreifbarkeitsregel* einführt: Eigenschaften von Objekten, die ein Objekt nicht selbst kennt, sind für das Objekt gleichgestellt mit denen von Objekten, die es zwar selbst kennt, auf die es aber nicht zugreifen darf.

68.2 Automatische Überprüfung

Man kann sich fragen, ob sich die Einhaltung des Law of Demeter so wie die Einhaltung der Zugreifbarkeitsregeln automatisch überprüfen lässt. Dabei gibt es aber ein Problem: Das Gesetz ist nämlich oben in Termini von Objekten, nicht von Variablen oder Typen formuliert. Eine Überprüfung würde also die Auswertung von konkreten Zuweisungen und damit des dynamischen Programmflusses erfordern, die aber mechanisch extrem aufwendig bis gar



nicht durchzuführen ist. Stattdessen prüfen automatische Checker des Law of Demeter zu meist lediglich die Variablen Deklarationen und ob alle Methodenaufrufe einer Klasse nur auf Ausdrücken erfolgen, die den Typ einer Instanzvariable oder eines formalen Parameters (wenn der Aufruf aus einer Methode heraus erfolgt, was meistens der Fall ist) haben. Daraus folgt, dass eine solche Prüfung in SMALLTALK nicht möglich ist (da Ausdrücke nicht typisiert sind).

Was aber tun, wenn man die Funktion von Ausdrücken wie oben haben und zugleich Demeters Gesetz folgen möchte? Die Antwort ist einfach: Man erweitert das Protokoll der Klasse des ersten Nachrichtenempfängers um die Methode(n), die man nicht verkettet aufrufen darf, also beispielsweise die Klasse des von **a** benannten Objekts um die Methode **doY**. Da **a** das Ergebnis von **doX**, nennen wir es **b**, irgendwo herhaben muss (sonst könnte es ja nicht zurückgeben), kann auch **a** die Methode **doY** aufrufen und das Ergebnis zurückgeben. Die Implementierung von **doY** würde dann durch **^ b doY** abgeschlossen.

Komplizierter wird die Sache jedoch, wenn die Verkettung länger ist, wenn also der zu vermeidende Ausdruck **a doX doY doZ** heißen würde, denn dann müsste auch noch **doZ** zur Klasse von **a** hinzugefügt und mit entsprechenden Implementierungen versehen werden. Man ahnt schon, wozu das führt: zu einem Wachstum des Protokolls von **a**.

68.3 Ein Beispiel

Ein konkretes Beispiel für die Verletzung des Law of Demeter ist die folgende alternative Implementierung der Methode **auswerten** in der Klasse **Klausel** (Zeilen 1650–1652):

```
1668 auswerten
1669   literale
1670     detect: [ :l |
1671       l negiert
1672         ifTrue: [l atom wert not]
1673         ifFalse: [l atom wert]]
1674       ifNone: [^ false].
1675     ^ true
```

Hier wird (in Zeilen 1672 und 1673) zunächst **atom** an **l** (ein Literal, das dem Empfänger der Methode, einer Klausel, bekannt ist) geschickt und dann an das Ergebnis, ein Atom, die Nachricht **wert**. Diese Verkettung ist ein Verstoß gegen das Gesetz Demeters. Die Musterlösung

```
1676 auswerten
1677   literale detect: [ :l | l auswerten] ifNone: [^ false].
1678   ^ true
```

enthält diesen Verstoß nicht. Dafür aber andere.



Selbsttestaufgabe 68.1

Versuchen Sie, bevor Sie weiterlesen, weitere Verstöße gegen das Gesetz Demeters in dem Beispiel zu finden.

Der erste Verstoß ergibt sich aus der Umsetzung der `:n`-Beziehung zwischen Klausel und Literalen über Zwischenobjekte. Eine Klausel kennt genaugenommen nicht ihre Literale direkt, sondern das Zwischenobjekt, in diesem Fall eine Instanz der Klasse `Set`. So stellt bereits der Aufruf von `atom` bzw. `auswerten` auf der Variable `l` eine Verletzung des Law of Demeter dar. Etwas deutlicher sieht man das, wenn man anstelle einer Menge und des Iterators `detect` ein Array und eine Zählschleife verwendet:

```
1679 1 to: literale size do: [ :i | (literale at: i) auswerten ...
```

Bei der Verwendung von Collections als Zwischenobjekte muss man also immer eine Ausnahme von Demeters Gesetz machen.

Der zweite Verstoß findet sich in Zeile 1672: `atom wert not` ist ein verketteter Ausdruck derselben Qualität wie `l atom wert` (selbe Zeile). Die Lösung wäre hier, der Klasse `Atom` eine Methode `not` zu spendieren, die den negierten Wert zurückliefert, aber warum das besserer Stil sein soll, ist kaum noch zu begründen. Auch wenn dieses Problem in JAVA und Co. nicht existiert (da hier die logische Negation keine Nachricht/Methode, sondern ein primitiver Operator eines primitiven Datentyps ist), so zeigt es doch die Grenzen des Law of Demeter auf. So ist das Gesetz auch nicht allgemein anerkannt, sondern umstritten; dennoch sollte man es verinnerlichen und sich bei Kettenausdrücken stets fragen, ob nicht eine Verlagerung einer hinteren Methode in eines der Objekte auf der Strecke sinnvoll wäre.

69 Klassenhierarchie

Der Begriff des Programmierstils kann weiter gefasst werden, als dies in den bisherigen Kapiteln dieser Kurseinheit der Fall war. Tatsächlich ist die Abgrenzung eines Stilbegriffs von allgemeinen Handlungsgrundsätzen und guter Praxis in der Programmierung nicht einfacher als in jeder anderen Disziplin, in der eine gewisse schöpferische Freiheit besteht — sie ist fast immer willkürlich. Im folgenden soll daher noch kurz ein Entwurfsprinzip vorgestellt werden, dass ich persönlich eher nicht als Stilfrage einstufen würde, dass aber dennoch häufiger in diesem Zusammenhang genannt wird.

Wie bereits in Kurseinheit 6 und (teilweise auch schon in Kurseinheit 2, Abschnitt 10.1) bemerkt, ist die Vererbung Aushängeschild und Problemkind der objektorientierten Programmierung zugleich. Es verwundert daher nicht, dass sich eine ganze Menge von Programmierrichtlinien mit genau diesem Thema beschäftigen. Die meines Erachtens wichtigste aller Regeln zu diesem Thema ist jedoch:



Mache alle Superklassen abstrakt.

Für JAVA lässt sich diese Regel auch als „deklariere jede Klasse entweder als abstrakt oder als final“ formulieren. Der Grund dafür, dass nur die Blätter der Klassenhierarchie instanzierbar sein sollen, ist einfach: Wenn man mit der Funktionalität einer Klasse (bzw. genauer und in diesem Fall wichtig, der Funktionalität der Instanzen einer Klasse) nicht zufrieden ist, will man die Implementierung der Klasse ändern. Als Programmiererin möchte man diese Änderung unabhängig von der Frage, ob davon auch andere Klassen betroffen sind, durchführen können. Deswegen wird man die Änderungen auch nur an der Klasse selbst und nicht etwa an einer ihrer Superklassen durchführen. Betrifft der Änderungswunsch eine geerbte Methode, so überschreibt man diese in der betreffenden Klasse nach seinen Vorstellungen. Nur wenn eine eingehende Analyse der Superklasse und all ihrer Subklassen ergibt, dass die gewünschte Änderung für alle sinnvoll ist und den Erwartungen der Klienten entspricht, kann man darüber nachdenken, die Änderung in der Superklasse durchzuführen.

Wenn nun aber die Klasse, deren Verhalten man ändern möchte, selbst Subklassen hat, dann ist man der Freiheit beraubt, nur für sich zu entscheiden — von jeder Änderung, die man durchführt, muss man fürchten, dass sie sich auf andere Klassen ausbreitet und den Vertrag dieser Klassen mit ihren Klienten bricht. (Siehe auch das Fragile-base-class-Problem in Kapitel 55). Etwas subtiler, aber genau dasselbe Problem, ereilt die Designerin von Klassenbibliotheken, wenn sie beschließt, das Verhalten einer Klasse zu ändern: Selbst wenn sie sich sicher ist, dass dies innerhalb der Bibliothek keine anderen als die gewünschten Auswirkungen hat, so kann sie doch nicht sicher sein, dass irgend eine Verwenderin ihrer Bibliothek von der Klasse, die sie gerade geändert hat, erbt und somit eine Verhaltensänderung erfährt, mit der sie nicht leben kann.

Nun ist aber, wie gerade erst (Kapitel 66) erwähnt, einer der wichtigsten Gedanken der Objektorientierung, existierenden Code, vor allem Bibliotheksklassen, per Vererbung wiederzuverwenden. Wenn die Bibliotheksdesignerin aber alle relevanten Klassen (das sind üblicherweise gerade die konkreten, also die instanzierbaren) final deklariert hat, dann ist das nicht möglich. Eine einfache Konvention erlaubt jedoch, diese Beschränkung zu umgehen: In der Bibliothek wird einfach die Klasse, von der geerbt werden soll, als abstrakt deklariert und per Vererbung eine Subklasse davon abgeleitet, die zunächst keine Änderungen (Differenzia) hinzufügt, dafür aber konkret (also instanzierbar) und auch final ist. Sollte die Bibliotheksdesignerin Änderungen durchführen wollen, kann sie das zunächst an ihrer finalen Klasse tun und nur, wenn sie sich vollkommen sicher ist, dass sie alle Klienten ihrer Bibliothek mit den Änderungen beglücken möchte, die Änderungen in der abstrakten Superklasse durchführen.



70 Fazit

Die Quintessenz der Bedeutung von Stil lässt sich kaum besser wiedergeben als durch ein weiteres Zitat von BERTRAND MEYER:

Good software is good in the large and in the small, in its highlevel architecture and in its low-level details. True, quality in the details does not guarantee quality of the whole; but sloppiness in the details usually indicates that something more serious is wrong too. ... A serious engineering process requires doing everything right: the grandiose and the mundane.

So you should not neglect the relevance of such seemingly humble details as text layout and choice of names. True, it may seem surprising to move on ... from ... formal specifications ... to whether a semicolon should be preceded by a space The explanation is simply that both issues deserve our care, in the same way that when you write quality O-O software both the design and the realization will require your attention.

Das Kapitel „A sense of style“ aus BERTRAND MEYERS Buch empfehle ich jeder, die glaubt, Stil sei eine persönliche Angelegenheit und ansonsten nicht so wichtig. Über MEYERS konkrete Vorgaben mag man streiten, über die prinzipielle Würdigung der Wichtigkeit von Stil hingegen nicht.

Wer JAVA programmiert, die mag sich die JAVA Coding Standards des Software Engineering Institutes der Carnegie-Mellon-Universität zu Gemüte führen. Man kann sich wundern, was an der Verwendung einer Programmiersprache alles regulierungsfähig ist (und fragen, warum man die eine oder andere Regel nicht in die Definition der Sprache gepackt hat). Lezenswert ist auch der Eintrag zum Thema „Programmierstil“ in der deutschsprachigen Wikipedia.



Link



WIKIPEDIA

71 Lösungen zu den Selbsttestaufgaben

Selbsttestaufgabe 68.1 (Seite 330)

Lesen Sie weiter!



Verzeichnis der Weblinks im Rand

- i Kurs 01613 www.fernuni-hagen.de/fu-search/index.jsp?query=01613
- iv Zugangshinweise für Datenbanken und elektronische Angebote www.ub.fernuni-hagen.de/datenbankenlieferdienste/zugangshinweise.html
Lesehilfe..... www.feu.de/ps/docs/K01814-1w.html
- 1 AC Kay „The early history of Smalltalk“ ACM SIGPLAN Notices 28:3 (1993) 69–95
..... doi.org/10.1145/154766.155364
- 3 Liste objektorientierter Programmiersprachen de.wikipedia.org/wiki/Liste_objektorientierter_Programmiersprachen
Simula..... de.wikipedia.org/wiki/Simula
Lisp de.wikipedia.org/wiki/Lisp
- 4 SL Tanimoto "A perspective on the evolution of live programming" in: Proc. of LIVE (2013) 31-34..... doi.org/10.1109/LIVE.2013.6617346
Y Shan „SMALLTALK on the rise“ Commun. ACM 38:10 (1995) 102–104
..... doi.org/10.1145/226239.226258
- 8 VisualWorks www.cincomsmalltalk.com/main/products/visualworks/
M Guzdial, E Soloway "Teaching the Nintendo generation to program" CACM 45:4 (2002) 17-21..... doi.org/10.1145/505248.505261
Squeak..... squeak.de/
- 9 Pharo pharo.org/
Smalltalk Express..... courses.cs.washington.edu/courses/cse341/99su/small-talk/newste.zip
Amber..... amber-lang.net/
AC Kay „The early history of Smalltalk“ ACM SIGPLAN Notices 28:3 (1993) 69–95
..... doi.org/10.1145/154766.155364
A Goldberg, D Robson: Smalltalk-80: The Language and its Implementation (Addison-Wesley 1983)..... stephane.ducasse.free.fr/FreeBooks/BlueBook/Bluebook.pdf
- 10 Stef's Free Online Smalltalk Books stephane.ducasse.free.fr/FreeBooks.html
- 12 Ockhams Rasiermessner de.wikipedia.org/wiki/Ockhams_Rasiermessner
- 27 Kurs 01665 www.fernuni-hagen.de/fu-search/index.jsp?query=01665
- 30 F. Steimann "None, One, Many - What's the Difference, Anyhow?" in: Proc. of SNAPL (2015) 294-308..... doi.org/10.4230/LIPIcs.SNAPL.2015.294
- 33 Mereologie..... de.wikipedia.org/wiki/Mereologie
- 41 Fluent API..... de.wikipedia.org/wiki/Fluent_Interface
- 62 Strukturierte Programmierung..... de.wikipedia.org/wiki/Strukturierte_Programmierung
E Dijkstra „Go-to statement considered harmful“ CACM 11:3 (1968) 147–148
..... doi.org/10.1145/362929.362947
- 67 Iterator..... de.wikipedia.org/wiki/Iterator
- 69 Kurs 01852 www.fernuni-hagen.de/fu-search/index.jsp?query=01852
- 73 A Black, N Hutchinson, E Jul, H Levy "Object structure in the Emerald system" in: OOPSLA (1986) 78–86..... doi.org/10.1145/28697.28706

- 75** Universalienstreit de.wikipedia.org/wiki/Universalienproblem
Extension und Intension de.wikipedia.org/wiki/Extension_und_Intension
- 76** Familienähnlichkeit de.wikipedia.org/wiki/Familienähnlichkeit
- 92** ISO/IEC 10027: Information technology -- Information Resource Dictionary System (IRDS) framework www.iso.org/standard/17985.html
- 96** Genus proximum et differentia specifica de.wikipedia.org/wiki/Genus_proximum_et_differentia_specifica
- 119** DHH Ingalls "A simple technique for handling multiple polymorphism" OOPSLA (1986) 347–349 doi.org/10.1145/28697.28732
- 132** Model View Controller de.wikipedia.org/wiki/Model_View_Controller
- 145** L Cardelli Type Systems" in: CRC Handbook of Computer Science and Engineering (1996) Chapter 103 homepage.divms.uiowa.edu/~tinelli/classes/185/Fall06/notes/cardelli-95.pdf
- 146** B Pierce: Types and Programming Languages (The MIT Press, 2002)
..... dblp.org/rec/books/daglib/0005958
- 155** Boolesche Algebra de.wikipedia.org/wiki/Boolesche_Algebra
Kurs 01661 www.fernuni-hagen.de/fu-search/index.jsp?query=01661
- 181** Kurs 01853 www.fernuni-hagen.de/fu-search/index.jsp?query=01853
- 195** J Palsberg, MI Schwartzbach: Object-Oriented Type Systems (John Wiley & Sons, 1994)
..... dblp.org/rec/books/daglib/0070753
- BC Pierce: Types and Programming Languages (MIT Press 2002)
..... dblp.org/rec/books/daglib/0005958
- 199** The Java Language Specification .. docs.oracle.com/javase/specs/jls/se11/html/index.html
The Long Strange Trip to Java www.blinkenlights.com/classiccmp/javaorigin.html
- 202** IEEE 754 de.wikipedia.org/wiki/IEEE_754
- 212** Oracle Java 8: Lambda Quick Start www.oracle.com/webfolder/technetwork/tutorials/obe/java/Lambda-QuickStart/index.html
- 217** Kurs 01853 www.fernuni-hagen.de/fu-search/index.jsp?query=01853
Java Pakete und Module docs.oracle.com/javase/specs/jls/se10/html/jls-7.html
- 220** zirkuläre Abhängigkeit en.wikipedia.org/wiki/Circular_dependency
- 221** Open Services Gateway Initiative.. de.wikipedia.org/wiki/OSGi
PS Canning, WR Cook, WL Hill, WG Olthoff "Interfaces for Strongly-Typed Object-Oriented Programming" in: Proc. of OOPSLA (1989) 457-467 doi.org/10.1145/74877.74924
- 222** Index.....
- 227** M Naftalin, P Wadler Java Generics and Collections (O'Reilly, 2006)
..... dblp.org/rec/books/daglib/0017180
- 230** M Torgersen, E Ernst, CP Hansen, P von der Ahé, G Bracha, N Gafter "Adding wildcards to the Java programming language" Journal of Object Technology 3:11 (2004) 97–116
..... doi.org/10.5381/jot.2004.3.11.a5
- 242** Kurs 01853 www.fernuni-hagen.de/fu-search/index.jsp?query=01853
F Steimann, J Öqvist, G Hedin "Multitudes of Objects: First Implementation and Case Study for Java" Journal of Object Technology 13:5 (2014) 1:1-33
..... doi.org/10.5381/jot.2014.13.5.a1

- 244** O Kiselyov, A Biboudis, N Palladinos, Y Smaragdakis "Stream fusion, to completeness" in:
Proc. of POPL (2017) 285-299 doi.org/10.1145/3009837.3009880
- 248** Threads in Java..... de.wikipedia.org/wiki/Thread_(Informatik)
- 249** Kurs 01853 www.fernuni-hagen.de/fu-search/index.jsp?query=01853
Kurs 01853 www.fernuni-hagen.de/fu-search/index.jsp?query=01853
- 256** C# Language Specification docs.microsoft.com/en-us/dotnet/csharp/language-refer-
ence/language-specification/index
- 258** Exception tunneling wiki.c2.com/?ExceptionTunneling
Versioning, Virtual, and Override.. www.artima.com/intv/nonvirtual.html
- 262** Kurs 01853 www.fernuni-hagen.de/fu-search/index.jsp?query=01853
- 269** B Stroustrup "A history of C++: 1979–1991" in: Second ACM SIGPLAN Conference on His-
tory of Programming Languages HOPL-II (1993) 271–297
..... doi.acm.org/10.1145/154766.155375
- 275** M Ellis, B Stroustrup: The Annotated C++ Reference Manual (Addison-Wesley 1990)
..... dblp.org/rec/books/law/EllisS90
- 276** J Coplien: Advanced C++ Programming Styles and Idioms (Addison-Wesley Professional 1991)
..... dblp.org/rec/books/law/Coplien92
- ECMA-Standard 367: EIFFEL: Analysis, Design and Programming Language (2nd edition, June
2006)..... www.ecma-international.org/publications/standards/Ecma-
367.htm
- B Meyer: Object-Oriented Software Construction (2. Ausgabe, Prentice Hall, 2000)
..... dblp.org/rec/html/books/ph/Meyer97
- 280** B Meyer "Overloading vs. object technology" Journal of Object-Oriented Programming
(Oct/Nov 2001) 3–7..... citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.1.9234
- 290** Kurs 01853 www.fernuni-hagen.de/fu-search/index.jsp?query=01853
- 291** B Meyer: Object-Oriented Software Construction (2. Ausgabe, Prentice Hall, 2000)
..... dblp.org/rec/html/books/ph/Meyer97
- 300** B Liskov, JM Wing "A Behavioral Notion of Subtyping." ACM Trans. Program. Lang. Syst. 16:6
(1994) 1811-1841..... doi.org/10.1145/197320.197383
- B Liskov, JM Wing "A Behavioral Notion of Subtyping." ACM Trans. Program. Lang. Syst. 16:6
(1994) 1811-1841..... doi.org/10.1145/197320.197383
- 304** W Codenie et al. „From custom applications to domain-specific frameworks“ CACM 40:10
(1997) 71–77 doi.org/10.1145/262793.262807
- 306** Kurs 01853 www.fernuni-hagen.de/fu-search/index.jsp?query=01853
L Mikhajlov, E Sekerinski "A study of the fragile base class problem" in: Proc. of ECOOP
(1998) 355–382 doi.org/10.1007/BFb0054099
- 307** S Williams, C Kindel „The Component Object Model: A Technical Overview“
..... www.cs.umd.edu/~pugh/com/
- 309** B Stroustrup: The Design and Evolution of C++ (Addison-Wesley 1994)
..... dblp.org/rec/books/daglib/0076736
- 312** W Wulf, M Shaw "Global variable considered harmful" ACM SIGPLAN Notices 8:2 (1973) 28–
34 doi.org/10.1145/953353.953355
- 313** C Ponder, B Bush "Polymorphism considered harmful" ACM SIGPLAN Notices 27:6 (1992)
76–79 doi.org/10.1145/130981.130991

- 315** F Steimann "Fatal Abstraction" in: Proc. of Onward! (2018) 125-130
..... doi.org/10.1145/3276954.3276966
- 322** Schlüssel und Fremdschlüssel [de.wikipedia.org/wiki/Schl%C3%BCssel_\(Datenbank\)](https://de.wikipedia.org/wiki/Schl%C3%BCssel_(Datenbank))
F Steimann "Content over container: object-oriented programming with multiplicities" in:
Proc. of Onward! (2013) 173-186 doi.org/10.1145/2509578.2509582
- 326** N Wirth: Algorithmen und Datenstrukturen (PASCAL-Version, Teubner Verlag, 2000)
..... dblp.org/rec/html/books/daglib/0095640
- 329** B Meyer: Object-Oriented Software Construction (Prentice Hall, 2000)
..... dblp.org/rec/html/books/ph/Meyer97
ungarische Notation..... de.wikipedia.org/wiki/Ungarische_Notation
- 332** Extract-method-Refactoring refactoring.com/catalog/extractMethod.html
- 336** Replace-conditional-with-polymorphism-Refactoring refactoring.com/catalog/replaceConditionalWithPolymorphism.html
KJ Lieberherr, IM Holland, A Riel "Object-oriented programming: An objective sense of style"
in: OOPSLA (1988) 323–334 doi.org/10.1145/62083.62113
- 341** B Meyer: Object-Oriented Software Construction (2. Ausgabe, Prentice Hall, 2000)
..... dblp.org/rec/html/books/ph/Meyer97
Programmierstil..... de.wikipedia.org/wiki/Programmierstil
CERT Oracle Secure Coding Standard for Java wiki.sei.cmu.edu/confluence/

Index

:	
:1-Beziehung	30
:n-Beziehung	30
A	
Abfrage	276
Abhangigkeit	217
abstrakte Klasse	s. Klasse
Abstraktionshierarchie	94
Accessor	53 , s. Zugriffsmethode
Aggregation	32, 283, 334
aktives Objekt	136 , 138
aktueller Parameter	45
Alias	18, 23 , 79, 168, 224, 283
Aliasing	15, 23 , 27, 52
Aliasing-Problem	314
Allgemeinbegriff	74, 313
Generalisierung als	96
nderungsabhangigkeit	217
Annotation	239, 247 , 261
Annotationstyp	199
Anweisung	13, 45
Application Programming Interface	198
Aristoteles	97
Array store exception	224, 230
Array-Initialisierer	199
Array-Literal	15 , 126
Array-Typ	199
Assoziativspeicher	124
atomares Objekt	129
Attribut	28
und Zustand	34
-wert	34
Attribute	261
Aufzhlungstyp	199
Ausdruck	36 , 37
Nachrichten-	37
primitiver	37
Zuweisungs-	37
Ausgabestrom	248
Auto boxing	201
Auto boxing und unboxing	262
Auto unboxing	201
B	
Base class	109
Basisklasse	302
Befehl	276 , 327
Behavioural subtyping	297
s. verhaltensbasiertes Subtyping	297
benannte Instanzvariable	29
Beziehung	312
:1- 30	
:n- 30	
leere	31
binre Methode	39
binre Nachricht	39
Block	58 , 266
boolesche Algebra	155
Boxing und Unboxing	264
Bytecode	197
C	
Call by reference	50 , 71, 201, 210, 269
in C#	255
Call by value	50 , 71, 201, 210, 268
Call stack	60
Change log	8
Child class	109
Client-Schnittstelle	103
Closure	60
Coding convention	329
Collection	
Garbage	s. Garbage collection
heterogene	187
-Klassen	120
Stream zum Zugriff auf eine	134
Common Intermediate Language	254
Common Type System	260
Compilation unit	4, 203, 218
Compilation units	197

Component Object Model	305
Continuation	60, 64
Cross cast	178, 236

D

Datenkapselung.....	276
declaration-site variance.....	265
Declaration-site variance	228
Default-Methode	222
Definition	151, 220
Deklaration.....	147, 151
deklariertes Element.....	146
Delegate.....	260, 261
Delegation.....	124, 270
denotationale Semantik	155
Dependent type.....	171
Dereferenzierung.....	320
Derived class.....	109
direkte Instanz	97, 99, 202
direkte Instanz	80
direkte Subklasse	108, s. Subklasse
direkte Superklasse	108, s. Superklasse
direkter Subtyp	172
Dispatch	217, s. a. dynamisches Binden
double	119, 209
multi.....	196
single.....	119
Double dispatch.....	119, s. Dispatch
Down cast	116, 178, 187, 227, 236, 251
Durchgängigkeit der Konzepte	2
dynamische Typprüfung.....	178
dynamischen Typprüfung.....	224
dynamisches Binden.....	49, 62, 64, 73, 145, 271, 291, 303

E

Eigenschaft	2, 75
Einfachvererbung	197
Endrekursion	66
Event	260
Event handler	260
Exception	
chaining	256
checked	245, 256
tunneling	256
unchecked	245, 256
Exception handling	
in Eifel	277

Expansion

rekursive.....	159, 180
rekursiver Typen	191

Export

dedizierter	277, 306
-------------------	----------

Extension	75, 80, 146
einer Generalisierung	97

F

Factory-Methode	89, 207
Fall through	214
Fallunterscheidung	119, 292, 309
gleiche	334
per dynamischem Binden	250, 286, s. a.
versteckte F.	
versteckte	49, 62
Familienähnlichkeit	76
Feld	28, 204
Filterfunktion	161, 164, 239
Fluent API	41, 42
formaler Parameter	23, 45, 46, 48, 125, 152
eines Blocks	61
formaler Typparameter	183
Forwarding	124, 270
Fragile-base-class-Problem	117, 256, 302
Fremdschlüssel	320
Friend	277
funktionale Anforderung	293
funktionale Programmiersprache	266, 319
funktionale Programmierung	3, 16, 18, 60

G

Garbage collection	5, 25, 145, 197, 254, 275
Geheimnisprinzip	36, 180, 316
Generalisierung	94, 173, 334
Ergebnis der	95
Vorgang der	95
Generics	183
generischer Typ	183
Genus et differentiae	97
Getter	53, 222, 257, 276
Lazy initialization per	87
Gleichheit von Objekten	16
globale Variable	21, 77
Goto	63

H

Hiding	21
Home context	59, 71, 116

bei Continuations.....	60
I	
Identität.....	2, 268
eines Objekts	12
von Objekten	16
Image.....	8
Impedance mismatch.....	320
imperative Programmierung.....	16
Implementation.....	57
Implementationsgeheimnis	36, 52, 55, 57, 86,
215, 306	
Indexer	29, 259
indirekte Instanz	98 , 202, s. Instanz
indirekter Subtyp	172
indizierte Instanzvariable	29
Information hiding.....	36, 276, 306, 316
Initialisierung	85, 100, 152
lazy.....	87, 131
für Klassenvariablen	89
Inklusionspolymorphie	177
Inlining	310
innere Klasse	203, 204, 318
Instanz.....	73, 79
direkte	80, 98
indirekte	80, 98
instanziierbar	
nicht.....	104
Instanziierung	79
einer parametrischen Typdefinition.....	188
eines parametrischen Typs.....	182, 184
Instanzmethode	84, 204
Instanzvariable.....	28, 76, 84, 204
integrierte Entwicklungsumgebung	198
Intension	75, 76, 146, 204
einer Generalisierung	96
Interface	55, 57, 148, 180
Marker.....	239
Tagging	239
Interface-als-Typ-Konzept	181, 302
interfacebasierte Programmierung.....	178, 181
Interfaceimplementation	220
Interfaceimplementierung	
explizite	262
implizite	263
Interfacetyp	199
Interfacevererbung.....	221
intern er Iterator	242
Invariante	169
Zustandswechsel-	300
Invarianz	175
Ist-ein-Beziehung.....	94
Iteration	63
externe	67
interne	67, 241
Iterator.....	240
J	
Java Virtual Machine	197
JUnit	115
K	
Kapselung	33, 36, 52, 313
Kardinalität	30
Klasse	
abgeleitete	109
abstrakte	101, 104, 106
als Interface	270, 272
in Eiffel.....	277
in Java	205
vs. Typ	178
Basis-.....	109
konkrete	105
Klassen	
-definition	76
-hierarchie	108
-methode	83
in Java	204
-typ	199
-variable	83
in Java	204
Klassenmethode	84
Klassenschnittstelle	313
Klassenvariable	84
Klassifikation	75, 79
Klonen	81
Kommentar	57
Komponente	318
Komposition	32, 33, 334
konkrete Klasse	105
Konstante	55
konstante Methode	55, 204
Konstruktor	84, 206
Kontext eines Blocks	58
Kontravarianz	175
von Wildcard-Typen	231

Kontrollfluß	
Beeinflussung durch Return	47
Kontrollstruktur	61
primitive	49
Kopie	
flache	129
tiefe	129
kovariante Redefinition	98
Kovarianz	175
von Wildcard-Typen	231
L	
Lambda-Ausdruck	60, 211, 241, 266
Laufzeittypfehler	224
Law of Demeter	334
Lazy initialization	87, s. Initialisierung
Lebenszyklus	26
Liskov-Substitutionsprinzip	298
Literal	13, 27, 55, 80
Klassen-	200
Literaloptimierung	71
Live programming	6
Live-Programmierung	4
logische Programmiersprache	319
lokale Variable	21, 28, 45, 152
Lokalitätsprinzip	307, 319
löschen von Methoden	122
M	
Marker interface	164
Marker-Interface	261
Mehrfachhierarchie	172
Mehrfachvererbung	270
Member	204, 220
Mereologie	33
Message dispatching	119
Message pattern	45
Metaklasse	82, 267
Metaprogrammiersystem	93
metasyntaktische Variable	38, 46, 203
Method ambiguous error	208
Method dispatching	119
Methode	45
binäre	191
konstante	20
Multi-	217
Methoden	
-aufruf	22, 42, 47, 48
-definition	45, 76
-kopf	45
-rumpf	45, 57, 152
-signatur	45, 57, 109
Methodendeklaration	152
Modul	215
Modularisierung	2
hierarchische	312
von Programmen	148
Monitor	246
Multi-dispatch	119, 209
Multi-Methoden	217
Multiplizität	30
N	
Nachricht	37
Nachrichtenausdruck	37
Nachrichtenkategorie	58
Nachrichtenselektor	39, 46
Nachrichtenversand	37
Name	17, 18
Namensäquivalenz	159
Navigation	320
Nebeneffekt	41
Nebenwirkung	41
Netzwerkmodell	27
nichtfunktionale Anforderung	293
nominale Typkonformität	
in Java	202
nominale Typkonformitätsdeklaration	221
Novarianz	175
von Java	202
Null pointer exception	150
Nur-Lesen	54
Nur-Schreiben	54
O	
Objekt	
atomares	13
strukturiertes	28
wachsendes	131
zusammengesetztes	14
Objektgeflecht	27
Objektorientierung	
klassenbasierte Form	73
prototypenbasierte Form	73
offene Rekursion	107, 116, 271, 303
Open Services Gateway Initiative	219

operationale Semantik	155	Repräsentationsobjekt	315
OSGi.....	219	Repräsentationsunabhängigkeit	276
Outer this	204	Return-Anweisung	44 , 47
P			
package local.....	239	bei Continuation	60
parallele Ausführung.....	62	Rolle	301, 329
parametrische Typdefinition	182	Rollenwechsel	
parametrischer Polymorphismus s. Polymorphismus		eines Objekts	178
Parent class.....	109	Routine	276
passives Objekt	136	Runtime type information	273
Pattern	260	S	
Persistenzlayer	320	Schlüssel	123, 320
Pipelining.....	242	Schlüsselwort	13, 63, 70
Platzhalter	19	Schlüsselwortnachricht	39 , 70
Pointer.....	19	Schnittstelle	148, 180
Pointervariable	19 , 25	eines Moduls	215
Polymorphie	49 , 176	Schnittstellenspezifikation eines Moduls	216
Inklusions-	s. Inklusionspolymorphie	Seiteneffekt	41, 42, 55, 214, 332
vs. Polymorphismus	49	self	48
Polymorphismus	49	semantischer Fehler	149 , 160
parametrischer	177, 182	Separation of concerns	313
vs. Polymorphie	49	Setter	53 , 222, 257, 276
primitive Methode	80, 85	Sichtbarkeit	20, 216
primitiver Typ	199	Sichtbarkeitsregeln	21
in Java	201	von Variablen	20
Programmierstil	329	Signatur	152
Property.....	257	Softwarekrise	2
Protokoll	48 , 55, 57 , 127, 153, 178	Speicherbereinigung	42, s. a. Garbage collection
Protokollbeschreibung	57	Spezialisierung	99 , 171, 334
Prototyp	76	Statik und Dynamik	
Pseudo-Typvariable Self	153 , 191, 193	Unterscheidung zwischen	11
Pseudovariable	14 , 23 , 45, 48, 70, 77	statischen Typprüfung	224
self und super im Home context	59	statisches Binden	49
Q			
qualifizierter Export	270, 277	Streams	241
R			
Raw type	236	String-Konkatenation	25
Redefinition	167 , 170, 277	Strings	14
Redundanz	148 , 149	Struktur	11
Refactoring	304	von Objekten	28
Referenz	19	Strukturäquivalenz	159
Referenzsemantik	19 , 27, 201, 223, 268	strukturierte Programmierung	63, 309
Reflection	247, 264	Subklasse	91 , 108
reflektiver Zugriff	219	direkte	108
relationales Datenmodell.....	27	in Java	205
		Subklassenbeziehung	108 , 205
		Substituierbarkeit	281, 292
		Subtyp	171
		Subtypenbeziehung	171
		Subtypenhierarchie	172

Subtyping	2, 172 , 291
verhaltensbezogenes	298
Subtyppolymorphie	177
Superklasse	108
direkte	108
Supertyp	171
Symbol	14
T	
Tagging interface	164
tatsächlicher Parameter	45 , 50
tatsächlicher Typparameter	183
Teil-Ganzes-Beziehung	32, 316
Template	272
temporäre Variable	41, 45
Test	148
Trennung der Belange	313
Tupel	289
Typ	146
annotation	148 , 149, 167
als Ausdruck einer Invariante	169
fehlerhafte	151
in Strongtalk	153
Weglassen von	
in Strongtalk	154
äquivalenz	159
nominale	159
strukturelle	159
erweiterung	162 , 205
expansion	264, 272
fehler	148 , 160
hierarchie	172
inferenz	147, 234, 264, 326
invariante	157
konformität	292
des Subtypen mit dem Supertypen	175
nominale	163
strukturelle	163
konstruktur	154
korrektheit	149
prüfung	
dynamische	150
regel	146
die von Ausdrücken einzuhalten sind	157
zur Zuordnung eines Typs zu Ausdrücken	
	157
system	146 , 157
optionales	145
-umwandlung	116, 178 , 227, 235, 236, 273
-variable	182
Type branding	160 , 161, 165
Type cast	273
Type erasure	236, 264
Typerweiterung	261, 281
Typinferenz	149
Typinvariante	148
Typisierung	156
typkonform	163 , s. a. Typkonformität
Typkonformität	281
U	
überladen	119 , 207 , 328
überschreiben	102 , 121, 167
in Eiffel	277
mit Aufruf von super	117
UML	<i>ii</i> , 30, 80
unäre Nachricht	39
ungarische Notation	263, 327
Unified Modeling Language	s. UML
Universalienstreit	75, 104
unsicher (unsafe)	254
unveränderliches Objekt	34
Up cast	178 , 236
Use-site variance	228
V	
Variable	2, 14, 17, 18 , 27
metasyntaktische	38
Variablendeklaration	149
veränderliches Objekt	34
verankerter Typ	283
Vererbung	2, 101, 261, 281
bei Typerweiterung	162
Vererbungsabhängigkeit	217
Vererbungsinterface	306
Vererbungsschnittstelle	103
Verhalten	11, 75
verhaltensbasiertes Subtyping	297
Verweis	19 , 27
Verweissemantik	19 , s. Referenzsemantik
Verwendung eines Programmelements	152
Virtual function table	272
W	
Wert	12 , 123
Werten	12

Wertsemantik	19 , 168, 201, 268	von Zuweisungen und Methodenaufrufen .	149
Wertzuweisung	2 , 21	Zu- <i>n</i> -Beziehung.....	30
Wiederverwendung	2	zusammengesetztes Objekt	129
Wildcard-Typ	229	Zustand.....	2 , 16 , 34 , 52
Wrapper-Typ	199	Zuweisung	21
Z		explizite.....	50, 286
Zeichenkette	14	implizite	22 , 50 , 51 , 165, 268
Zu-eins-Beziehung	30	linke und rechte Seite	22
Zugreifbarkeitsregel	335	unzulässige.....	22
Zugriffsmethode ... 53 , 220, s. a. Getter und Setter automatische Einrichtung von	93	Zuweisungskompatibilität	157 , 158 , 163, 205, 281, 291
Zugriffsmodifikator	180 , 216	bei parametrisierten Subtypen	186
Zulässigkeit		unter parametrischem Polymorphismus.....	182
von Methodenaufrufen	158	Zuweisungsoperator	21 , 23 , 41
von Zuweisungen	158	Zuweisungsversuch	286